The Dissertation Committee for Amol Pramod Nayate
certifies that this is the approved version of the following dissertation:

Transparent Replication

Committee:

Mike Dahlin, Supervisor

Lorenzo Alvisi

J. C. Browne

Arun Iyengar

Emmett Witchel

Transparent Replication

by

Amol Pramod Nayate, B.S., M.A.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2006

Dedicated to my wife, parents, and brothers

Acknowledgments

I would first and foremost like to thank my advisor, Mike Dahlin. Mike has consistently done more than what is required of an advisor and helped me in many ways through my graduate life. His interest in the subject matter and enthusiasm to pursue research is inspiring and contagious. Mike also has excellent insight into systems problems and has always had the right advice for me whenever I have gotten stuck. Mike is the reason I am interested in systems research. I hope to continue working with Mike throughout my professional career.

I have also been very lucky to work with Arun Iyengar, another incredibly brilliant and inspiring individual. I have been working with him since I was a summer intern at IBM and will continue working with him there. Like Mike, Arun has taken an active interest in my career and professional development and I have learned an unbelievable amount from him.

I would like to thank Lorenzo for valuable discussions. Other than Mike and Arun, no one understands my work to the level that Lorenzo does. Lorenzo, like Mike, is also an excellent lecturer and puts a lot of effort into teaching.

I would like to thank J. C. Browne and Emmett Witchel for serving on my PhD committee. I have a deeper understanding of the subject thanks to

their insight and suggestions. My dissertation is much better thanks to their help and supervision.

I would like to thank students with whom I have closely worked: Bharat Chandra, Lei Gao, Nalini Belaramani, Jiandan Zheng, Arun Venkataramani, and Praveen Yalagandula. We have always formed great teams from having spent long nights in the office together. I would like to thank the students with whom I don't have any co-authored publications but have had extensive discussions: Jian Yin, Jean-Philippe Martin, Rama Kotla, Ravi Kokku, and Mitul Tewari.

I could not have made it through graduate school without support from friends in the department - Prem Melville, Joseph Modayil, Mikhail Bilenko, Aniket Murarka, etc. - and other friends in our LASR group - Jeff Napper, Navendu Jain, Sugat Jain, Jairam Mudigonda, Taylor Riche, Dimitry Kit, Prince Mahajan, Upendra Shevade, Ajay Mahimkar, Alan Clement, Harry Li, and Eunjin Jung.

My parents and brothers have always been there for me and have been a constant source of support and encouragement for me throughout my life. I wouldn't be here were it not for them. I am forever greatful for what they have always done for me.

Finally, I would like to thank my wife Anagha, who has been unbelievably supportive throughout the whole process. She has been there with me through my entire dissertation writing and job search process and has been the

reason I was able to get through those successfully. I can't thank her enough.

AMOL NAYATE

The University of Texas at Austin

December 2006

Transparent Replication

Publication No. _____

Amol Pramod Nayate, Ph.D.
The University of Texas at Austin, 2006

Supervisor: Mike Dahlin

Increasing user expectations and demands have caused the evolution of web services away from single-server systems and toward distributed systems for their ability to provide *improved throughput*, *improved availability* and *reduced response times*. However, for a service to run on a distributed system, each running instance must be able to access data that are shared among the instances. Although existing off-the-shelf *replication systems* - e.g. distributed file systems [52, 61, 32, 38, 41], replicated databases [64, 75], distributed hash tables [58, 59, 63, 34], etc. - simplify access to shared data by exporting well-researched interfaces, their implementations are typically not engineered for the unique environments presented by many web services. For example, replication systems that require synchronization across multiple nodes to handle modified data [38, 12] or systems that require all nodes to keep a copy of all data [64, 75] may not be practical for use in such services.

Although the problem of general replication is not possible to solve [11, 62, 33] we focus our study on a class of single-writer services that we denote

Information Dissemination Services that form a restrictive but important set of web services.

Our research makes two key contributions. First, we show that for a class of single-writer services that we denote *Information Dissemation Services* TRIP replicates dynamic data in a manner that is nearly transparent to the service. We (1) develop a novel dual-channel replication algorithm for TRIP that utilizes spare network background traffic to speculatively replicate data in a safe, non-interfering fashion, (2) show how to integrate safe speculative replication with mechanisms that use invalidates to provide consistency, and (3) demonstrate how our combination of consistency and safe speculative replication allows us to provide near-ideal consistency, performance, and availability for Information Dissemination Services.

Second, we show that the core principles behind building TRIP can be extended to build a new replication framework and more general replication toolkit. In particular, we show that it is possible to extend our dual-queue mechanisms developed for TRIP to a multi-writer environment where nodes can synchronize multiple incoming streams of data and consistency information. Our extension allows providing various forms of consistency for arbitrary topologies - two key properties provided by the PRACTI [6] (Partial Replication, Arbitrary Consistency, Topology Independence) architecture.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Increasing user expectations and demands have caused the evolution of web services away from single-server systems and toward distributed systems for three reasons. First, a distributed system results in *improved throughput*, because the computational load of the running service can be spread across multiple nodes. Second, a distributed system yields *improved availability* because when a node fails or becomes disconnected from the network its clients can contact another working, reachable node for service. Finally, a distributed system yields *reduced response times* because an instance of the service can be configured to run nearer to the user.

However, for a service to run on a distributed system, each running instance must be able to access data that are shared among the instances. Although existing off-the-shelf *replication systems* - e.g. distributed file systems [52, 61, 32, 38, 41], replicated databases [64, 75], distributed hash tables [58, 59, 63, 34], etc. - simplify access to shared data by exporting well-researched interfaces, their implementations are typically not engineered for the unique environments presented by many web services. For example, to provide strong consistency guarantees, AFS [38] requires global synchroniza-

tion between all nodes that replicate a particular piece of data when that piece of data is published [12]. For applications that can tolerate reduced consistency, such an operation may be prohibitively expensive in an environment where nodes are large in number or connected over slow networks. Similarly, Bayou [64] requires that all nodes keep a complete copy of all shared data. In an environment where the shared data tend to be large, dynamic, or shared over a slow network, such a requirement may be impractial.

Therefore in this dissertation we address building *TRIP*, a new replication system built for web services. We do not advance the state of the art in replication interfaces; instead, we aim to build a replication system that exports existing interfaces.

Our research makes two key contributions. First, we show that for a class of single-writer services that we denote *Information Dissemation Services* TRIP replicates dynamic data in a manner that is nearly transparent to the service. We (1) develop a novel dual-channel replication algorithm for TRIP that utilizes spare network background traffic to speculatively replicate data in a safe, non-interfering fashion, (2) show how to integrate safe speculative replication with mechanisms that use invalidates to provide consistency, and (3) demonstrate how our combination of consistency and safe speculative replication allows us to provide near-ideal consistency, performance, and availability for Information Dissemination Services. Furthermore, we show that our architecture vastly simplifies reasoning about replication details such as consistency.

Second, we show that the core principles behind building TRIP can be extended to build a new replication framework and more general replication toolkit. In particular, we show that it is possible to extend our dual-queue mechanisms developed for TRIP to a multi-writer environment where nodes can synchronize multiple incoming streams of data and consistency information. Our extension allows providing consistency for arbitrary topologies - two key properties provided by the PRACTI [6] architecture.

1.1 Challenges

The emergence of web services places unique demands that guide our design of TRIP. We focus on five key challenges:

1. *Heavy data sharing*: Each replica of a web service typically replicates a large subset of the total data used by the service. Therefore, for example, we consider replication systems that retain per-file (or per-object) information about which file is replicated at which node [38, 41] to be unsuitable for web services.

2. *Data volatility*: For web services that serve dynamic content, any change to the data must be reflected at all nodes that replicate such data. Thus, high data volatility can cause the system to expend large amounts of network resources to maintain data consistency.

3. *Environment heterogeneity*: As web service replication environments become increasingly heterogenous, replication systems face new challenges

functioning across (1) networks with varying bandwidth, reliability, and latency behavior and (2) possibly mobile devices with varying power, storage space, and network connectivity.

4. *Simplicity*: Although hardware costs continue to decrease, the cost of developer time does not. Therefore, to reduce the time needed to develop a system we require that a replication system provide simple, well defined semantics of its behavior.

5. *CAP Dilemma*: The *CAP* dilemma identified by [11, 33, 62] and formalized by [33] proves that in a network that is prone to partitions, it is not possible to provide both strong consistency and complete availability. Therefore, any replication system must make a fundamental trade-off between the two.

1.2 Transparent replication

In an ideal world, the interface provided by our replication system would be *transparent* to the application. We define transparency as the ability of the replication system to provide at each node the abstraction of a local, non-replicated storage system. A key requirement of transparency is that it may not introduce new behaviors into the system. Therefore, a transparent replication system would allow a developer to build and test a service for a single server and automatically deploy the service at multiple replicas by transparently replicating the shared data. We speculate that a transparent

replication system could thus vastly simplify development, testing and deployment of a service.

To provide transparent replication, we require four properties from a hypothetical replication system: consistency, availability, performance, and resource non-interference:

- *Consistency*: An ideal replication system must guarantee that (1) for any view of the data exposed to the application by the replication system, there must exist a sequence of events that is valid according to semantics of the application that could have generated that view of the data, and (2) the application should not see data that are significantly stale in real-time.

- *Availability*: An ideal replication system would ensure that any access to replicated data by the application should be satisfiable.

- *Performance*: An ideal replication system would ensure that the response time of accessing replicated data would be comparable to that of accessing local storage.

- *Resource non-interference*: An ideal replication system would mask its usage of system resources from the service. For example, a running service should not have to compete for network bandwidth from the replication system. The goal of transparent replication is thus to hide any anomalous behavior that can result from replicating data over a potentially slow network that is prone to partitions.

Unfortunately, meeting all of the these goals in the general case is impossible. For example, the CAP dilemma proves that in a network that is prone to partitions it is impossible to provide consistency and complete availability simultaneously. Therefore instead of searching for a general solution to replication we (1) focus our attention on a small but important set of services, *Information Dissemination Services*, that tolerate relaxed consistency and availability and (2) do not attempt to provide complete transparency. In this dissertation we build and evaluate a novel replication system, *TRIP*, and show that it provides an accetable degree of transparency despite the CAP dilemma.

1.3 Approaches

We describe our approach in two broad steps. First, we describe *TRIP*, a replication toolkit that provides nearly transparent replication for Information Dissemination Services. Second, we show that some of the insights that we gain from the design of TRIP can be extended to more general replication scenarios. In particular, we show that the mechanisms built for TRIP can be combined with other mechanisms to build a replication system whose functionality subsumes that of many existing existing replication systems.

1.3.1 TRIP

TRIP attempts to partly circumvent the CAP dilemma by exploiting three key characteristics of the target environment. First, since Information

Dissemination Services have only one writer, we show that providing FIFO consistency is sufficient to ensure that applications observe a sequentially consistent view of data. Furthermore, Information Dissemination Services allow for relaxed availability semantics because the write operation is completely available at all but one node. Second, we note that many distributed web services can tolerate some degree of staleness in their data - e.g. a non-critical news site can serve data that are a few minutes stale - allowing TRIP to transparently relax real-time consistency requirements without affecting service-visible semantics. Finally, because in most situations the system has more bandwidth than is required, we utilize spare bandwidth to agressively prefetch data for availability and performance.

TRIP is built using three key mechanisms: Separation of data and metadata, synchronizing data and metadata streams for consistency, and pushing data in priority order along a low-priority network channel:

1. *Separation of data and metadata*: The origin server performs a write operation by sending (1) metadata (lightweight *invalidate* messages) on a FIFO-ordered channel and (2) updated data on a separate channel.

2. *Synchronizing data and metadata streams for consistency*: Each TRIP receiver contains a novel scheduler that (1) provides *sequential consistency* by processing each invalidate message in FIFO order and delaying a data message until its corresponding invalidate message has been pro-

cessed, and (2) enforces Δ-coherence by processing each received invalidate message within Δ units of time of its arrival.

3. *Pushing data in priority order along a low-priority network channel*: We use existing mechanisms [67, 45, 8, 9] to build network channels that send data only over spare bandwidth. To maximize availability and performance, TRIP uses a lossy priority queue to order unsent data such that data that have a higher likelihood of being requested at a receiver are sent ahead of other data.

Although the separation of data and metadata has been used extensively in existing literature [38, 41, 61], they typically enforce consistency by requiring the sender to send data and metadata messages over a single FIFO channel. Our novel architecture's ability to separate the data and metadata streams and synchronize them at the receiver for consistency is vital to enabling safe speculative replication because it allows the receiver to treat the data stream as a lossy, asynchronous, unordered data channel and allows the sender to send data in an order that maximizes performance and availability at the receiver. We discuss this design in further detail in chapter 3 and show in chapter 6 that this architecture can be extended to support more services than Information Dissemination Services.

1.3.2 PRACTI

In ongoing joint work, *PRACTI* [6], we construct a replication toolkit that builds on the insights of TRIP and extends them to more general environ-

ments. In particular, PRACTI (1) supports applications beyond single-writer Information Dissemination Services (e.g. multi-writer applications), and (2) allows for nodes to communicate in any arbitrary topology instead of restricting communication to a tree topology. PRACTI provides three key properties: *Partial Replication (PR)*, which refers to the ability of the replication system to allow any node to store any subset of the total data present in the system, *Arbitrary Consistency (AC)*, which refers to the ability of the replication system to support providing a broad range of consistency semantics, and *Topology Independence (TI)*, which refers to the ability of the replication system to allow any pair of nodes in the system to synchronize updates.

We extend our three key ideas in TRIP - separation of data from metadata, synchronizing data and metadata streams at the receiver, and prefetching data on a background channel in order of priority - to provide PRACTI properties for multi-writer applications through three key observations. First,we note that TRIP provides some degree of Partial Replication because although it requires all nodes to observe all metadata it does not require nodes to replicate all data. Second, we note that by controlling the policy for scheduling data and metadata messages at the receiver it is possible to configure the receiver to provide a broad range of consistency semantics (Arbitrary Consistency). Finally, we note that although TRIP does not directly provide Topology Independence, the mechanisms that TRIP uses to send data and metadata can be extended to use over an arbitrary topology.

We add three key mechanisms to TRIP to enable TRIP to provide

PRACTI properties: log-based metadata exchange, imprecise invalidates, and a more flexible consistency scheduler. First, to provide Topology Independence, we use existing peer-to-peer log propagation protocols [64, 75] to store only metadata in logs and utilize alternate channels to propagate data over an arbitrary topology. Second, to enable proper Partial Replication, we introduce *imprecise invalidates* that summarize a series of metadata messages in an efficient but compact representation to allow nodes to receive only summaries of metadata for data that they do not replicate. Finally, to enable Arbitrary Consistency we add to the TRIP scheduler's ability to schedule data and metadata messages the ability to control what subset of locally-replicated data is visible to the overlying service.

1.4 Contributions

Our work makes three key contributions. First, we describe *TRIP*, a novel replication algorithm that provides nearly transparent replication for Information Dissemination Services by (1) separating data and metadata paths, (2) employing a novel scheduler that synchronizes streams of data and metadata to meet consistency guarantees at a receiver, and (3) speculatively pushing data over a low-priority channel in an order that maximizes performance and availability. Second, we evaluate the performance of TRIP using both simulation and prototype results and show that TRIP outperforms several other replication strategies for Information Dissemination services. Finally, we show that TRIP's rules to synchronize streams of data and metadata at the receiver

to provide consistency can be extended to support a much larger set of services than Information Dissemination Services.

We envision two key benefits of our work. First, we expect that service developers that use our TRIP toolkit could significantly improve the time they spend building and debugging Information Dissemination Services. Second, we envision that maintainers of existing information dissemination services may be able to gain more efficiency or robustness from their system by replacing their current replication system with TRIP.

The rest of the dissertation proceeds as follows. We provide background on our work in chapter 2 and describe the design of the TRIP algorithm in chapter 3. Chapter 4 describes the structure of the simulator that we use, and chapter 5 provides evaluation results yielded by our simulator. The 6 describes the implementation of the TRIP prototype on PRACTI and evaluates the prototype. Finally, we present related work in 7 and conclude in chapter 8.

Chapter 2

Background

2.1 Information Dissemination Services

We focus our research on providing transparent replication for *Information Dissemination Services*, a small but important set of web services. Such services have the property that (1) all writes originate at a single server that we denote the *origin* server, and (2) *edge servers* or *replicas* only read data. Edge servers assemble fragments, cache data, etc. to generate web pages.

2.2 System Model

Figure 2.1 provides a high level view of the environment we assume. An *origin server* and several *replicas* (also called content distribution nodes or edge servers) share data, and logical *clients*—either on the same machine or another—access the service via the replicas, which run service-specific code to dynamically generate responses to requests [3, 4, 13, 27, 65, 69]. The system typically uses some application-specific mechanism [14, 40, 73] to direct client requests to a good (e.g., nearby, lightly loaded, or available) replica. The design of such a redirection infrastructure is outside the scope of the paper; instead, we focus on the *data replication middleware* that provides shared state

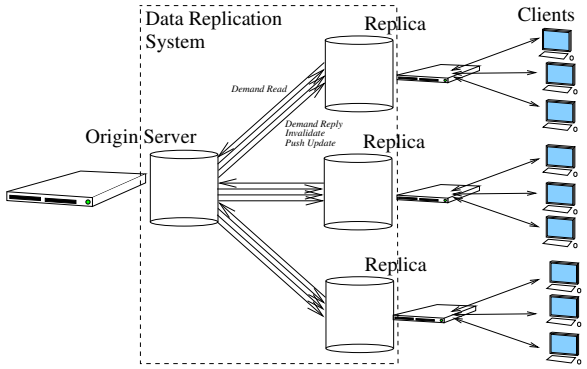

Figure 2.1: High level system architecture.

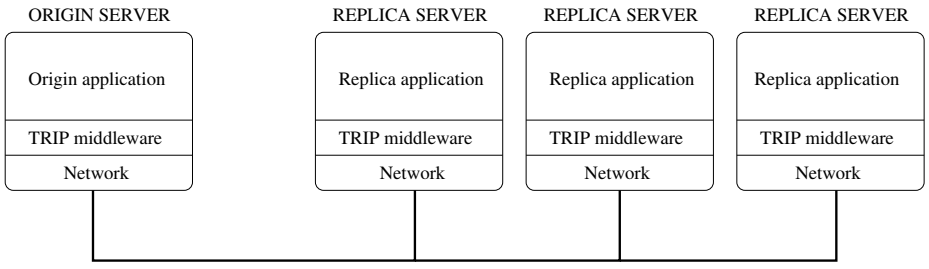

Figure 2.2: Lower level system architecture.

across the origin server and replicas. Figure 2.2 illustrates the logical position of the replication middleware; the TRIP middleware component running at the origin intercepts write calls and transparently disseminates data to the corresponding components at the edge servers.

Proposed service replication architectures [3, 4, 13, 27, 65, 69] vary in their assumptions about the number of replicas (e.g., 10 replicas to thousands), whether a given replica is typically installed for long periods of time on the same machine(s) or whether replicas are dynamically created, destroyed, or moved over fine time scales to respond to changing demand, and whether a replica caches a small subset of hot data or replicates most or all of a service. We focus on supporting on the order of 10 to 100 long-lived replicas that each have sufficient local storage to maintain a local copy of the full set of their service's shared data. Our protocol remains correct under other assumptions, but optimizing performance in other environments may require different trade-offs.

2.3 Consistency and timeliness

This study focuses on protocols that simultaneously enforce both sequential consistency, which restricts the permitted ordering among reads and writes across all objects, and Δ-coherence, which limits the real-time duration between when a write of an object occurs and when the write becomes visible to subsequent reads. The rest of this section defines these concepts more precisely.

Evaluating the semantic guarantees of large-scale replication systems requires careful distinctions between *consistency*, which constrains the order that updates across multiple memory locations become *observable* [30] to nodes in the system, *coherence*, which constrains the order that updates to a single location become observable but does not additionally constrain the ordering of updates across different locations, and *staleness*, which constrains the real-time delay between when an update completes and when it becomes observable. Adve discusses the distinction between consistency and coherence in more detail [2].

To support transparency, we focus on providing sequential consistency. As defined by Lamport, "The result of any execution is the same as if the [read and write] operations by all processes were executed in some sequential order and the operations of each individual processor appear in this sequence in the order specified by its program." [48] Sequential consistency is attractive for transparent replication because the results of all read and write operations are consistent with an order that could legally occur in a centralized system, so—absent time or other communication channels outside of the shared state—a program that is correct for all executions under a local model with a centralized storage system is also correct for a distributed storage system.

Typically, providing sequential consistency is expensive in terms of latency[51, 12] or availability[11]. However, we restrict our study to *dissemination services* that have one writer and many readers, and we enforce *FIFO consistency* [51] under which writes by a process appear to all other processes

in the order they were issued, but different processes can observe different interleavings between the writes issued by one process and the writes issued by another. Note that for applications that include only a single writer, FIFO consistency is identical to sequential consistency or the weaker causal consistency.

Although ensuring sequential consistency at each replica provides strong semantic guarantees, clients accessing a service through the replicas may observe unexpected behaviors in at least two ways due to communication channels outside of the shared state.

First, because sequential consistency does not specify any real-time requirement, a client may observe a stale version of the service. For example, if a network partition separates a replica from the origin server, the view of the service provided by the replica will not reflect recent updates even if the view continues to obey sequential consistency. A user could observe, for example, the anomalous behavior of a stock price not changing for several minutes during a disconnection. In this case, physical time acts as a communications channel outside of the control of the data replication middleware that could allow a user to detect anomalous behavior introduced by the replication system.

Therefore, we allow systems to enforce timeliness constraints on data updates by providing Δ-*coherence*, which requires that any read reflect at least all writes that occurred before the current time minus Δ. By combining Δ-coherence with sequential consistency, TRIP enforces a tunable staleness limit on the sequentially consistent view. The Δ parameter reflects a per-

service trade-off between availability and worst case staleness: reducing Δ improves timeliness guarantees but may hurt availability because disconnected edge servers may need to refuse a request rather than serve overly-stale data.

Second, some redirection infrastructures [14, 40, 73] may cause a client to switch between replicas. Even if each replica provides a sequentially consistent view of the data, a client switching between replicas may see inconsistencies. For example, consider two replicas r_1 and r_2 where r_2 processes messages somewhat more slowly than r_1. If objects A and B are initially in states A_0 and B_0, then A is written to state A_1, and finally B is written to state B_1, a client could read object B and observe state B_1 from replica r_1 and then switch to replica r_2 and read object A and observe state A_0. Even though neither r_1 nor r_2 observes any state inconsistent with the notion that A_1 *happens before* [47] B_1, by switching between replicas the client can observe such an inconsistent state. In Section 3.5 we discuss how to adapt Bayou's session consistency protocol [64] to our replication environment to ensure that each client observes a sequentially consistent view regardless of how often the redirection infrastructure switches the client among replicas.

Chapter 3

TRIP Design

In this chapter we discuss the design of the TRIP algorithm. We utilize three key mechanisms to build TRIP: (1) Separating data and metadata paths, (2) synchronizing these streams of data and metadata at the receiver to provide consistency, and (3) pushing data in order of *priority* along a low-priority network channel.

TRIP is based on a novel replication algorithm that revolves around two simple parts: (1) the origin's self-tuning efforts to send updates in priority order without interfering with other network users and (2) each replica's efforts to buffer messages it receives, to apply them in an order that meets consistency constraints, and to delay applying some of these messages to improve availability and performance.

3.1 Design choice

We make our design decisions based on 3 key requirements. First, to provide performance and availability, we make it our goal to (1) keep the cache as full as possible by prefetching unseen updates, and (2) keep more benefitial data in the cache over less benefitial data

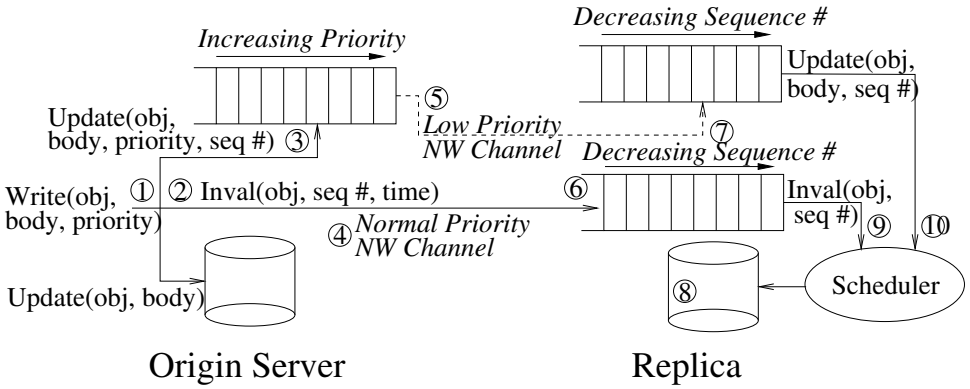

Figure 3.1: Overview of replication algorithm. The circled numbers are discussed in the text.

Second, to provide resource non-interference, we use a low-priority network channel to send updates.

Third, to provide sequential consistency we build an additional channel along which we send metadata that allows the receiving replica to order updates to restrict applications to observe a sequentially consistent view of data.

3.2 Algorithm

Figure 3.1 provides a high-level view of the algorithm for synchronizing a replica's data store with the origin server's. When the origin server writes an object (number ① in the figure), it immediately sends an invalidation to each replica ② and it enqueues the body of the update in a priority queue for each replica ③. In contrast with the immediate transmission of invalidations

on a normal-priority lossless network connection ④, each priority queue drains by sending its highest-priority update to its replica via a low-priority network channel when the network path between the origin server and replica has spare capacity ⑤.

At the replica *both* invalidation ⑥ and update ⑦ messages that arrive are buffered rather than being immediately applied to the replica's local data store ⑧. A scheduler at each replica applies invalidations in strict sequence-number order ⑨, delaying the application of each successive invalidation until its corresponding update appears in the update buffer or until its deadline (under Δ-coherence) arrives. Similarly, when the scheduler at a replica applies a buffered update ⑩, it always applies the one with the lowest available sequence number and it only applies an update if all invalidations with lower sequence numbers have already been applied.

The full algorithm must also handle demand reads, network disconnections, and machine failures. We therefore detail the server and replica algorithms in the next two sections. Then Section 3.5 discusses several limitations of the basic algorithm and possible optimizations available within this framework.

3.3 Origin server

The core of the origin server is a novel and generally-applicable architecture for push-based prefetching where each update channel to a replica consists of a priority queue of updates that drains via a low-priority network

Algorithm 1 Origin server

```
State
  seqNo; // Global sequence number
  storage; // Seq number + body of each object
  nReplicas; // Number of replicas
  updtChnl[]; // Lossy, prior. order, low prior. link
  invDemChnl[]; // Lossless, FIFO channels
Write(objID, body, priority, timestamp):
  seqNo++;
  storage.update(objId, body, seqNo);
  for (i = 0; i < nReplicas; i++) do
    invDemChnl[i].send(INVAL, objId, seqNo, timestamp);
    updtChnl[i].insert(UPDATE, objId, body, seqNo, priority);
  end for
On receiving (READ, objId) from replica:
  (body, objSeqNo) = storage.get(objId);
  invDemChnl[replica].send(REPLY, objId, body, objSeqNo);
  updtChnl[replica].cancel(objId);
```

connection to a replica. By combining a priority queue and a low-priority network protocol, the updates' channel provides for self-tuning prefetching for each replica. When the network between the origin server and a replica provides a large amount of spare bandwidth, the priority queue drains quickly and the channel approximates a lossless, FIFO, push-all channel. But, when network bandwidth is scarce, only valuable items are sent and the buffering delay allows multiple updates of the same data to collapse into a single update and save network bandwidth [5]. Note that unlike many traditional prefetching protocols [24, 35, 36, 56, 68], there is no pre-set threshold that determines whether a given object is valuable enough to send; instead, TRIP relies on the low-priority network protocol to ensure that objects are only sent when the value of doing so exceeds the cost [42].

In order to integrate sequential consistency and Δ-coherence with self-tuning updates, the origin server separates each replica's invalidation channel

from its update channel. When an update occurs, the origin server immediately sends the invalidation to each replica, but it enqueues the update bodies in the per-replica priority queues. Unfortunately, separating these channels prevents replicas from depending on message arrival order for consistency, so the origin server associates a sequence number with each update and each stored object, and it includes an object's sequence number in all invalidation, update, and demand-reply messages.

Algorithm details. As we show in the pseudocode in Algorithm 1, the origin server maintains a global monotonically increasing sequence number *seqNo*, local *storage* with the body and sequence number of each object, per-replica channels *invDemChnl[]* for sending invalidations and demand replies, and per-replica channels *updtChnl[]* for pushing updates.

The algorithm proceeds as follows. To write an object, an origin server increments *seqNo*, updates *storage* with *seqNo* and the object's new body, sends invalidations on each replica's *invDemChnl*, and enqueues updates on each replica's *updtChnl*.

Each enqueued update includes a *priority* that specifies the update's relative ranking to other pending updates. Our interface allows a server to use any algorithm for choosing the priority of an update, and this paper does not attempt to extend the state of the art in prefetch prediction policies. A number of standard prefetching prediction algorithms exist [24, 35, 36, 56, 68] or the server may make use of application-specific knowledge to prioritize an item

(e.g., a news editor may know that the day's headline article will be widely read before the system has measured the story's read frequency). Note that some implementations may extend this interface to specify different priorities for propagating a given update to different replicas to, for example, account for different access patterns at different replicas.

When the origin server receives a demand *read(objId)* from a replica, it retrieves from its local store the object's body and per-object sequence number, and it sends on the replica's *invDemChnl* a demand reply message. Notice that this reply includes the sequence number stored with the object when it was last updated, which may be smaller than the current global *seqNo*. Upon sending a demand reply to a client, the origin server also cancels any push of the object to that client still pending in the *updtChnl* for the receiving replica.

Communication channels. The system design depends on the distinct properties of the *invDemChnl*s and the *updtChnl*s.

Each *invDemChnl* for invalidations and demand replies is a lossless FIFO channel that operates at normal network priority. Our protocol uses a persistent message queue [39] to ensure that this channel is lossless even across crashes and network partitions, which dramatically simplifies crash recovery.

Each *updtChnl* provides an abstraction suited for self-tuning push-based prefetch by (1) buffering updates in a priority queue and (2) sending them across the network using a low priority network protocol. Three actions manipulate each per-replica priority queue. First, an *insert* adds an update with

a specified priority. If another update to the same *objId* occupies the priority queue, the older update is discarded. An implementation may bound the upper size of the priority queue buffer and discard low priority items to maintain this size bound. Second a *cancel(objId)* call removes any pending update for *objId*. Third, a worker thread loops, removing the highest priority update from the queue and then doing a low-priority network send of a push-update message containing the *objId*, *body*, and *seqNo* of the item. The low priority network protocol should ensure that low priority traffic does not delay, inflict losses on, or take bandwidth from normal-priority traffic; a number of such protocols have been proposed [8, 9, 55, 67, 45]. To ensure that the bulk of unsent updates remain in the priority queue where they remain eligible to be replaced by later updates, network sends should block once a limited local network buffer fills. The local network protocol buffer should therefore be large enough to support good sustained throughput, but no larger.

3.4 Replica

The core of each replica is a novel *scheduler* that coordinates the application of invalidations, updates, and demand read replies to the replica's local state. The scheduler has two conflicting goals. On one hand, it would like to delay applying invalidations for as long as possible to minimize the amount of invalid data and thereby maximize local hit rate, maximize availability, and minimize response time. On the other hand, it must enforce sequential consistency and Δ-coherence, so it must enforce two constraints:

C1 A replica must apply all invalidations with sequence numbers less than N to its storage before it can apply an invalidation, update, or demand reply with sequence number N. [1]

C2 A replica must apply an invalidation with timestamp t to its storage no later than $t + \Delta - maxSkew$.

Δ specifies the maximum staleness allowed between when an update is applied at the origin server and when the update affects subsequent reads, and *maxSkew* bounds the clock skew between the origin server and the replica.

Each scheduler applies invalidations in sequence number order and maximizes the amount of valid data in its local storage by trying to delay applying an invalidation with sequence number N until it has an update with the same sequence number. A scheduler is forced to apply an invalidation earlier than that in two circumstances: (1) the staleness deadline for an invalidation expires or (2) a demand read reply that reflects state M ($M > N$) arrives at the replica, forcing the scheduler to immediately apply pending invalidations with sequence numbers up to M to avoid stalling the demand read.

Algorithm details. The pseudocode in Algorithm 2 describes the behavior of a replica. Each replica maintains five main data structures. First, a replica maintains a local data store *storage* that maps each object ID for the shared state to either the tuple *(INVALID, seqNo)* if the local copy of the object is

[1] We show that enforcing condition C1 yields sequential consistency in the Appendix.

Algorithm 2 Replica

```
State
  storage; // Validity, sequence number, and body of each object
  pendingInval; // Received but unprocessed invalidation
  pendingUpdate; // Received but unprocessed updates
  delta; // Max staleness between server and replica
  maxSkew; // Max clock skew between server and replica
On receiving (INVAL, objId, seqNo, timestamp) on invDemChnl:
  pendingInval.put(objId, seqNo, timestamp);
On receiving (UPDATE, objId, body, seqNo) on updtChnl:
  pendingUpdate.put(objId, body, seqNo);
If pendingUpdate.head.seqNo≤pendingInval.nextSeqToProcess():
  // Scheduler applies an update
  (objId, body, seqNo) = pendingUpdate.removeHead();
  if (seqNo ≥ storage.getSeqNo(objId)) then
    storage.update(objId, VALID, seqNo, body);
  end if
  if (seqNo == pendingInval.nextSeqToProcess()) then
    pendingInval.doneProcessing(seqNo);
  end if
If currentTime() ≤ pendingInval.head.timestamp+delta-maxSkew:
  Scheduler applies an invalidate
  applyNextInval(); // See below
On local call to read(objId):
  if (VALID == storage.getState(objId)) then
    return storage.getBody(objId);
  end if
  send(READ, objId) to origin server;
  storage.waitUntilValid(objId);
  return storage.getBody(objId);
On receiving (REPLY, objId, body, seqNo) on invDemChnl:
  while (pendingInval.nextSeqToProcess() ≤ seqNo) do
    applyNextInval(); // See below
  end while
  storage.update(objId, VALID, seqNo, body); // Unblock read
applyNextInval() // Internal private method called from above
  (objId, seqNo, timestamp) = pendingInval.readHead();
  if (seqNo ≥ storage.getSeqNo(objId)) // 'At least once' chnl then
    storage.update(objId, INVALID, seqNo);
  end if
  pendingInval.doneProcessing(seqNo);
```

in the invalid state or the tuple *(VALID, seqNo, body)* if the local copy of the object is in the valid state. Second, a replica maintains *pendingInval*, a list of pending invalidation messages that have been received over the network but not yet applied to *storage*; these invalidation messages are sorted by sequence number. Third, a replica maintains *pendingUpdate*, a list of pending pushed updates that have been received over the network but not yet applied to the local data store; notice that although the origin server sorts and sends these update messages by priority, each replica sorts its list of pending updates by *sequence number*. Finally, Δ specifies the maximum staleness allowed between when an update is applied at the origin server and when the update affects subsequent reads, and *maxSkew* bounds the clock skew between the origin server and the replica.

Scheduler actions. After *INVAL* and *UPDATE* messages arrive and are enqueued in *pendingInval* and *pendingUpdate*, a scheduler applies these buffered messages in a careful order to meet the two constraints above and to minimize the amount of invalid data.

The scheduler removes the update message with the lowest sequence number from its *pendingUpdates* and applies it to its *storage* as soon as it knows it has applied all invalidations with lower sequence numbers from *pendingInvals*. Applying a prefetched update normally entails updating the local sequence number and body for the object, but if the locally stored sequence number already exceeds the update's sequence number, the replica must dis-

card the update because a newer demand reply or invalidation has already been processed. Also note that in the case where update N arrives before invalidation N is applied, update N can be applied as soon as invalidation $N-1$ has been applied and then invalidation N need never be applied. In this case, the procedure informs the *pendingInval* queue that *seqNo* has been processed, which allows *pendingInval* to garbage collect the message and to acknowledge processing of invalidation *seqNo* to the origin server.

The scheduler removes the invalidation message with the lowest sequence number from *pendingInval* and applies it to its *storage* when the invalidation's deadline arrives at $timestamp + \Delta - maxSkew$. The *pendingInval* queue and network channel normally provide FIFO message delivery, and they guarantee at least once delivery of each invalidation when crashes occur. To support end-to-end at-least-once semantics, before applying an invalidation, a replica verifies that it is a new one, and after applying an invalidation a replica calls *pendingInval.doneProcessing(seqNo)* to allow garbage collection of the message and to acknowlege processing of invalidation *seqNo* to the origin server.

Processing requests from clients. When servicing a client request that reads object $objId$ (either as input to a dynamic content-generation program or as the reply to a request for a static data file), a replica uses the locally stored body if $objId$ is in the $VALID$ state. But, if the object is in the $INVALID$ state, the replica sends a demand request message to the server and then waits

for the demand reply message. Note that by sending demand replies and invalidations on the same FIFO network channel, the origin server guarantees that when a demand reply with sequence number N arrives at a replica, the replica has already received all invalidations with sequence numbers less than N, though some of these invalidations may still be buffered in *pendingInval*. So when a demand reply arrives, the replica enforces condition C1 by simply applying all invalidation messages whose sequence numbers are at most the reply's sequenceNumber before applying the reply's update to the local state and returning the reply's value to the read request.

Our protocol implements an additional optimization (not shown in the pseudo-code for simplicity) by maintaining an index of pending updates searchable by object ID. Then, when a read request encounters an invalid object, before sending a demand request to the origin server, the replica checks the pending update list. If a pending update for the requested object is in this list, the system applies all invalidations whose sequence numbers are no larger than the pending update's sequence number, applies that pending update, and returns the value to the read request.

A remaining design choice is how to handle a second read request r_2 for object o_2 that arrives when a first read request r_1 for object o_1 is blocked and waiting to receive a demand reply from the origin server. Allowing r_2 to proceed and potentially access a cached copy of o_2 risks violating sequential consistency [2] if program order specifies that r_1 *happens before* r_2. On the other hand, if r_1 and r_2 are issued by independent threads of computation

that are not synchronized, then the threads are logically concurrent and it would be legal to allow read r_2 to "pass" read r_1 in the cache [48, 30]. TRIP therefore provides two options. *Conservative* mode preserves transparancy but requires a read issued while an earlier read is blocking on a miss to block. *Aggressive* mode compromises transparency because it requires knowledge of application internals, but it allows a cached read to pass a pending read miss. Our experiments examine this trade-off in more detail.

Operating during disconnection. When a replica becomes disconnected from the server due to a network partition or server failure, the replica attempts to service requests from its local store. If the local copies of most objects are valid, a replica may be able to mask the disconnection for an extended period. Note that to enforce Δ-coherence, a replica must block all reads if it has not communicated with the origin server for Δ seconds. We use a heartbeat protocol to ensure liveness when the network is available. But, if a read miss occurs during a disconnection, it logically blocks until the connection is reestablished and the server satisfies the demand miss.

In a web service environment, blocking a client indefinitely is an undesirable behavior. Therefore, TRIP provides three ways for services to give up some transparancy in order to gain control of recovery in the case where a replica blocks because it is disconnected from the origin server.

First, after a time-out a read can return an error code to the calling edge server program. Although a correct program should always check for

error codes on file or database reads, in practice this interface is not fully transparent because (a) many applications fail to check for error codes on IO operations and (b) the actions an application should take on a read error may differ in this distributed case (where, say, redirecting the request to a different replica may work) versus the centralized case (where probably little can be done.)

Second, rather than require applications to deal with time-outs internally, TRIP can be configured to take two actions when a demand read times out: (1) signal the redirection layer [14, 40, 73] to stop sending requests to this replica and (2) signal the local web server infrastructure to close all existing connections to all clients and to respond to subsequent client requests with an HTTP redirect [26] to a different replica. The approach then relies on client-initiated request retransmission for end-to-end recovery [11]. This option provides less precise control to the application, but it also requires less invasive modifications of the service-specific code.

Third, given the choice between reducing availability and increasing staleness during disconnections, some services may choose the latter. Such services may configure TRIP to increase Δ when it detects a disconnection from the server. This increase allows the system to further delay applying pending invalidations and thus maximize the amount of valid local data and maximize the amount of time the replica can operate before suffering a miss. For example, if a replica sets $\Delta = \infty$ during disconnections, it will apply no invalidations while disconnected, but it may serve arbitrarily stale data.

3.5 Limitations and optimizations

Our current protocol is limited in at least two ways. These limitations could be addressed with future optimizations.

First, as described in Section 2.3 our current protocol can allow a client that switches between replicas to observe violations of sequential consistency. Therefore, for best results the redirection algorithm should direct a client to the same replica for long periods of time.

We speculate that a system could adapt Bayou's session guarantees protocol [64] to maintain sequential consistency semantics when a client switches replicas. In particular, a replica's web server could insert an HTTP cookie reflecting the highest sequence number observed by a client in responses to a client and inspect this cookie on all requests from a client. If the sequence number in a request exceeds the replica's sequence number, the replica web server signals the replication infrastructure to process pending invalidations to bring the sequence number to a point where the request can be processed. This optimization compromises transparency, but we speculate that the necessary modifications to the server would generally not be too invasive.

Second, our protocol sends each invalidation to all replicas even if a replica does not currently have a valid copy of the object being invalidated. We take this approach for simplicity and because we primarily target environments that trade cheap bandwidth and storage for improved availability and responsiveness and where replicas are therefore able to maintain valid copies

of most data. Our protocols could be extended to more traditional caching environments where replicas maintain small subsets of data by adding callback state [38] that filters the invalidations sent to each channel to include only invalidations for objects that have been demand fetched or prefetched since the objects previous invalidation. Given our target environment, we have no current plans to pursue this optimization.

3.6 Transparency

To meet our goal of providing *transparent* replication, we aim to meet four key goals:consistency, availability, performance, and resource non-interference. We discuss the impact of each of these requirements on our algorithm design below:

Consistency We provide *sequential consistency* along with Δ-coherence by (1) requiring the origin server to serialize all writes, (2) requiring replicas to apply invalidate messages in order within Δ units of time of sending, and (3) disallowing replicas to apply data messages that allow clients to observe any inconsistency in data.

Availability TRIP provides availability benefits by aggressively prefetching data and delaying invalidates at a replica to (1) ensure that the local storage is as full as possible at any time, and (2) objects are received in priority order to ensure that more useful objects are chosen to be kept over less useful ones

(resource utilization), and (3) taking appropriate action when it detects a network partition - e.g. by increasing the value of Δ.

Performance Other than TRIP's attempt to keep a replica's local storage as valid as possible, TRIP improves system performance in two other ways: (1) When a read is blocked for an update that is also blocked for an invalidate, TRIP accelerates applying invalidate messages until the blocked read is satisfied, and (2) TRIP uses a regular priority channel to push objects when they are sent as the result of a demand fetch.

Resource non-interference TRIP enforces resource non-interference in two manners. First, TRIP protects network resources by prefetching updates over a low-priority channel[67, 45]. Second, TRIP spares storage resources at the origin server by allowing the origin server to discard any update scheduled to be transmitted.

We note that a shortage of storage resources at the origin server can be harmful in environments where the system has sufficient bandwidth to push all updates, because a discarded update will cause the system to fail to function like push-all and cause the replica to demand-fetch data.

3.7 Δ trade-off

Figure 3.2 demonstrates the trade-off created by the Δ parameter between consistency on the one hand and performance and availability on the

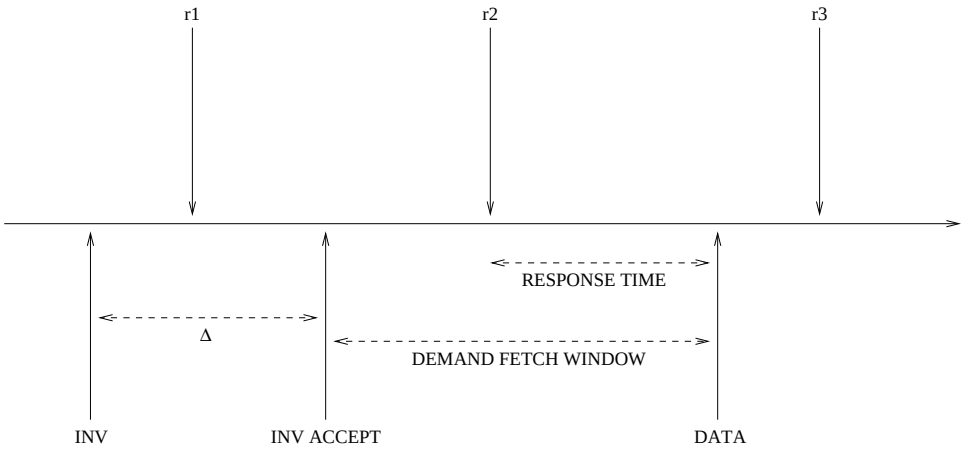

Figure 3.2: Effect of Δ on performance, availability, and consistency

other. We assume that all events shown in the figure are for the same object. The figure shows three requests: $r1$, $r2$, and $r3$, such that (1) request $r1$ arrives before the invalidate is accepted and is hence satisfied using locally-stored data, (2) request $r2$ arrives after the invalidate is accepted and hence must wait for the arrival of data (either due to prefetching or an explicit demand fetch) and (3) request $r3$ arrives after the object has arrived and therefore observes fresh data immediately.

Our figure illustrates precisely the problem that occurs due to our separation of data and metadata paths. The duration between when an invalidate for an object arrives and when the data arrives creates a fragile window of time (denoted as the demand-fetch window in the figure) when any request for an object could force the replica to demand-fetch the object. A request that is blocked for the arrival of an object may observe high response times -

and hence low performance - or it may observe data unavailability in the event that a network partition prevents the arrival of the object. Each request in our figure illustrates a different situation; $r1$ exhibits low consisteny but good performance and availability, $r2$ observes high consistency but low performance and availability, and $r3$ observes both high consistency and high performance and availability.

3.8 TRIP variation: replica-driven prefetching

We briefly discuss a variation of TRIP where all data prefetching in controlled by replicas. In particular, (1) each replica maintains its own priority queue of objects that have been invalidated but not prefetched, (2) the origin server sends with each invalidate message the *priority* of the associated write, and (3) each replica continuously sends requests for objects to be prefetched over low priority.

This variation of TRIP has three key benefits. First, this variation of TRIP may significantly reduce the load on the origin server because it frees the origin server from maintaining potentially large per-replica priority queues. Second, because a replica is best suited to choose a prefetching policy[24], this variation of TRIP allows each replica to use complex, location-specific prefetching policies. Finally, this variation of TRIP allows each replica to make its own trade-off with respect to storage resources; for example, this variation allows a replica to choose to restrict the size of its priority queue at the cost of potentially lost opportunities to prefetch data.

We note, however, that this variation of TRIP may suffer reduced performance or availability compared to the standard version of TRIP. In particular, when a piece of data changes the origin must wait for the replica to prefetch it. In the context of figure 3.2, this variation would yield large *Demand-fetch windows* because the origin server may have to wait one round-trip time between when it first writes an object and when the replica makes a request for it.

3.9 Discussion

Our TRIP architecture of logically separating the stream of invalidates from the stream of updates vastly simplifies the process of reasoning about consistency. For many existing systems, proving their correctness often requires sophisticated tools (e.g. Teapot [17], TLA [49], etc.) and potentially considerable effort on the part of the user [16]. Our architecture demonstrates that consistency can be provided by (1) accepting invalidates in an order dictated by simple, locally-enforceble rules and (2) restricting an application to observing only those data that are consistent according to the order specified by such invalidates. The key observation that reasoning about consistency does not require reasoning about the order in which data arrive at the receiver allows us to treat consistency as a safety property and vastly simplifies the task of proving consistency properties. We note, however, that to ensure that data are available at the receiver (liveness) it is important for the sender to send fresh data when required.

We illustrate the simplicity of proving our consistency safety property in our proof in the Appendix. We write a hand-generated proof that proves the following safety property:

[For Information Dissemination Services] condition C1 provides sequential consistency.

We note that we do not prove that TRIP provide Δ-coherence for two key reasons. First, we note that when a replica is partitioned from the network it is impossible to provide timeliness guarantees [11, 33]. Second, we note that when the bandwidth available to the system is not sufficient to guarantee that an object will reach a replica within Δ units of time, it is not possible to enforce Δ-coherence. Therefore, we provide coherence on a best-effort basis only.

Chapter 4

Simulator

Our simulator is written as an event-driven simulator that chooses a model of the environment that is (1) simple to simulate, and (2) models only bottleneck components.

This chapter proceeds as follows. We first describe the various algorithms that our simulator currently supports in section 4.1. Second, we outline our simulation model as well as our assumptions and simulation limitations in section 4.2. Third, we describe the component of our simulator responsible for generating and parsing requests in section 4.3. Finally, we describe the core simulator in section 4.4.

4.1 Algorithms

We currently build our simulator to support three data-dissemination algorithms other than TRIP. Although simple, these algorithms are used in actual implementations and hence are used as a basis for comparison to TRIP:

- *Push All*: The Push All algorithm disseminates data by requiring the origin server to push all updates to all replicas in FIFO order[64, 75]

- *Demand Only*: The Demand Only algorithm requires the origin server to handle writes by pushing invalidate messages to all replicas but requires replicas to fetch objects on demand[38, 41]. No data are prefetched to replicas.

- *Threshold-based*: The Threshold-based algorithm requires the origin server to handle writes by dynamically choosing to push either updates or invalidates to replicas. We compute an expected probability p that an object will be requested at a replica before it is modified, and choose a threshold k such that the origin server pushes the object to all replicas if $p > k$ or sends only invalidate messages for that object.

4.2 Model

We make several simplifying assumptions about the environment that we simulate:

1. We assume that the network bottleneck is near the origin server; therefore, rather than modeling separate origin-replica network links, we assume that the origin server has a shared outgoing network link along which it sends data to all replicas.

2. For simplicity, we simulate the origin server and each replica to use a single thread such that each server handles only a single request at a time.

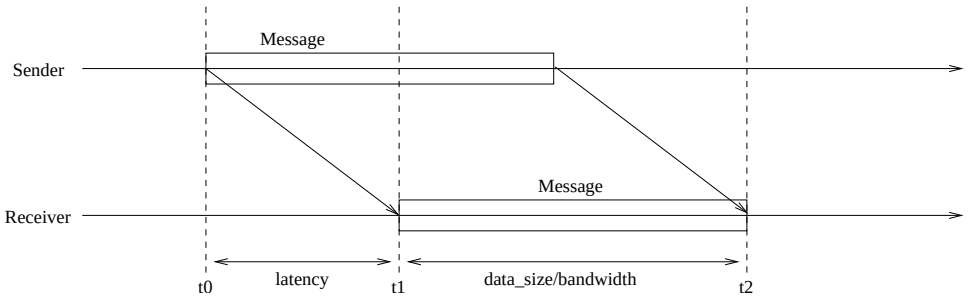

Figure 4.1: Time taken to send and process a message

3. We assume that each client that communicates with a replica has a very high-bandwidth connection with that replica. Therefore we assume that the replica consumes no network resources to satisfy client requests that request locally stored valid data.

4. We model each local computation at the origin server or replica to consume no resources. In particular, we assume that a server is occupied only when it is sending or receiving data. For web services that make heavy use of resources other than the network - e.g. CPU or disk - our model may not be appropriate.

5. We assume that nodes are organized in a star topology such that all replicas communicate only with the origin server.

4.2.1 Network

Figure 4.1 shows our model of simulating network messages. We assume that (1) the network adds a certain size-independent *latency* to transmit

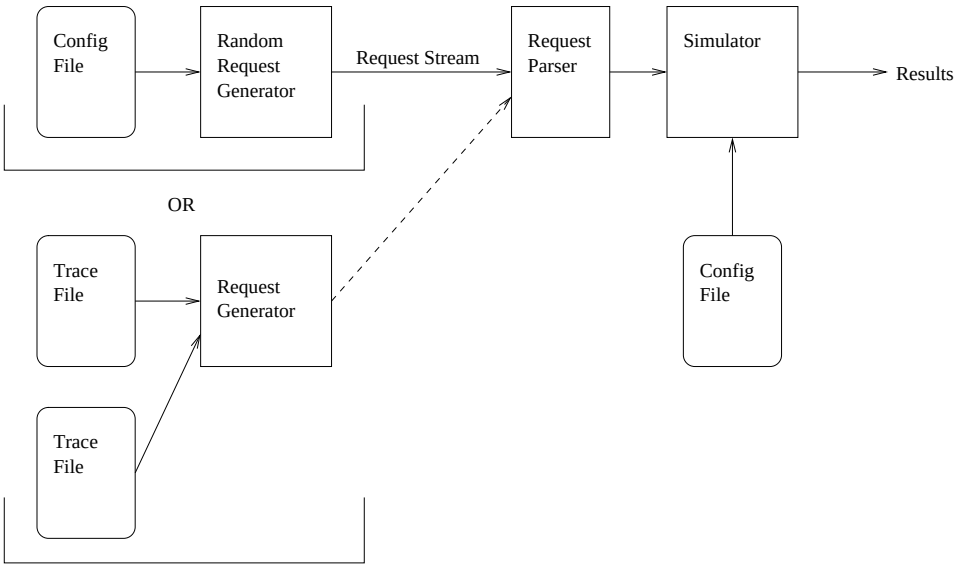

Figure 4.2: High level component view of the simulator.

a message, and (2) each message occupies both the network and the CPU at the sender and receiver for $data\_size/bandwidth$ units of time. For simplicity, we do not model networks that allow senders or receivers to transmit multiple messages simultaneously. Thus, the time spent between when the sender starts sending a message and it is fully received at the receiver is given by $latency + data\_size/bandwidth$. Although we assume that the bandwidth does not change throughout the simulation, we compute a separate *latency* for each message. Our current implementation computes a random latency for each message that is within 90% of a user-chosen $mean\_latency$.

Time	Operation	File Number	Replica ID
104.38	READ	90	C
104.42	READ	7	D
104.50	WRITE	24	A

Table 4.1: Example table of requests generated by the RequestGenerator.

4.3 Request generator/parser

As shown in figure 4.2, two components are responsible for generating and parsing requests: the *RequestGenerator* and *RequestParser*. Internally. the simulator uses a simulator-specific, human-readable tracefile that is generated by the RequestGenerator and read by the RequestParser. The RequestParser then represents the internal tracefile as populated data structures readable by the core simulator component.

A RequestGenerator may either generate synthetic requests itself or serve as a translator that reads one or more existing trace files as input and generates a simulator-specific merged tracefile as output. Our separation of the RequestGenerator from the rest of the system therefore allows us to abstract away the format of the original tracefiles1. We define the format of the internal tracefile and describe our RequestGenerators below.

Internal trace-file The format of the RequestGenerator output stream is shown in table 4.1 and consists of a time-ordered list of requests that take place throughout the simulation. The first field represents the time at which the request occurred. The second field represents the type of the request, or the

operation commited by the request. A read request corresponds to an HTTP GET request, and a write corresponds to the modification of a file. The third field indicates the number of the file on which the request occurred. Although a real application would use file names (such as *index.html* or *figures/image.gif*), using numbers makes the simulator more efficient. We do not expect that our use of file numbers rather than names is a significant restriction because for any original tracefile that uses file names it is possible to build a RequestGenerator that maps them to numbers. Finally, the fourth field indicates the ID of the replica at which the request occurred. This field is a string, although for practical simulations it is expected to contain an IP address. The first request from the figure, thus, would translate to *Read request for file 90 at replica C at time 104.38*. In the internal trace file, these fields are in a tab-separated order, with each request on a single line.

Note that this format of the trace file allows for any replica to write to a file, although Information Dissemination Services only permit writes to occur at the origin server. We build our tracefile format to allow for the presence of multiple writers for extensibility; for our simulations of Information Dissemination Services we designate the single writer with replica ID "A" in the trace.

4.4 Simulator core

The core of the simulator is built as an event-driven state machine that orders and processes both events that arrive on the request stream and

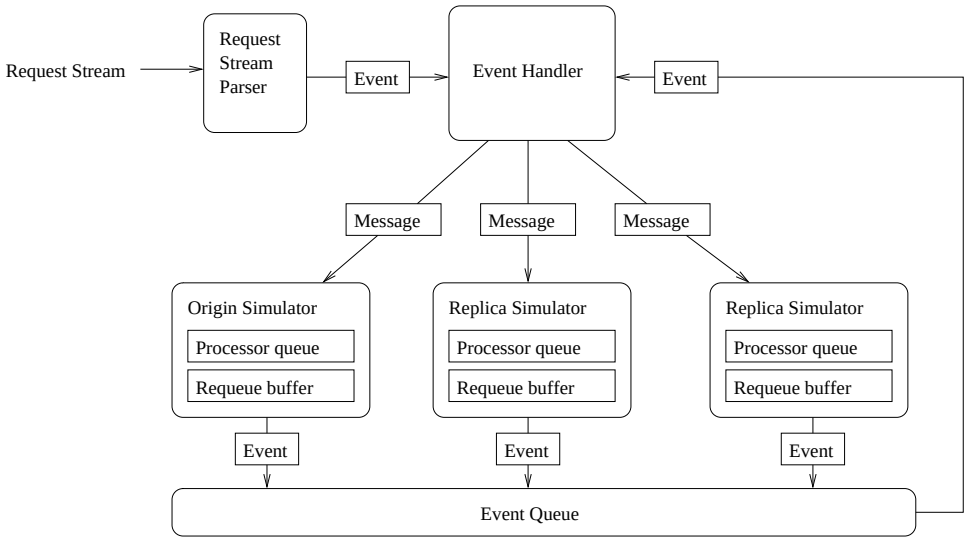

Figure 4.3: Low level component view of the simulator.

internally-generated events. Figure 4.3 shows a component view of the simulator configured for two replicas. We describe each component below.

4.4.1 Event

The *Event* data structure represents any system event and currently includes the following:

1. NULL: Used for debugging purposes

2. GENSTAT: The GENSTAT event causes the system to generate simulation statistics. This event occurs at least once at the end of simulation, but can be generated periodically to print statistics at various intervals throughout the simulation.

3. PROC: The PROC event contains a replica ID and indicates that the replica with the given ID has finished its current task. The Event Handler then assigns a new task for that replica.

4. MESSAGE: This event contains a source and target replica ID to indicate that the target replica has received a message. We note that since we only simulate Information Dissemination Services that communicate in a star topology, we disallow messages that are sent from one replica to another.

4.4.2 Message

This *Message* data structure is stored with a MESSAGE event and represents any transfer of data on the underlying network. Currently, we support the following types of messages:

1. NULL: Used for debugging purposes

2. READ_REQUEST: Represents a request either from a client to its replica or from a replica to the origin server to fetch a given object.

3. WRITE_REQUEST: Represents a replica's request to write to an object.

4. READ_FINISH: Represents a reply to a read request. This message may be sent either by the origin server to a replica or by a replica to the client that made the original request.

5. WRITE_FINISH: Represents a reply to a write request.

6. INVALIDATE: Represents an invalidate message sent by the origin server.

7. PUSH_FILE: Represents an object that is pre-emptively pushed by the origin server to a replica.

We make the key simplification where only two classes of messages - *READ_FINISH* and *PUSH_FILE* - have non-zero sizes, since they represent file transfers. For simplicity, we assume that *data_size* is 0 for all other messages; therefore, such messages occupy neither the sender nor the receiver and reach their destination in *latency* units of time. We speculate that our assumption does not influence results significantly because (1) we expect that messages that transfer files account for most of the total bandwidth consumed, and (2) in a realistic implementation we expect all control messages to fit within one network packet[22].

Note that we currently model *WRITE_REQUEST* messages to be of size 0, although in practice they would be accompanied by new data and thus have a non-zero size. We made this simplification for three reasons. First, this assumption helps vastly simplify the simulator implementation. Second, our implementation of the TRIP prototype co-locates the single writer and the origin server; therefore, our simulator models our actual implementation of TRIP. Finally, the cost of sending write messages is the same independent of the data replication algorithm that we employ; hence, although our assumption may influence our quantitative results, we expect that our simulator

would reflect correct trends when used to make relative comparisons across algorithms.

4.4.3 Origin/Replica simulators

The Origin and Replica simulators implement the various data dissemination algorithms that we simulate. We build extensibility into our simulator by defining standard interfaces for OriginSimulators and ReplicaSimulators such that supporting a new algorithm involves only writing a new OriginSimulator and ReplicaSimulator for that algorithm. The interface that we design for these allows them to be notified when they must handle either a PROC event or a MESSAGE event.

Each replica and origin simulator has two key local data structures: a *processor queue*, and a *requeue buffer*. Each message that arrives gets placed on the processor queue for processing. When the replica or origin simulator finishes the current task - as is indicated by the arrival of a PROC event - the relevant ReplicaSimulator or OriginSimulator will dequeue the next message from its processor queue and process it. The processor queue is implemented as a simple FIFO queue.

The requeue buffer allows a replica to exploit parallelism when the ReplicaSimulator is written to support multiple clients. We note that as described in chapter 3, our algorithm can be run in a reduced-consistency, agressive mode where replicas assume that all requests that they receive in parallel are independent. Since under such a configuration we allow any read for a

locally-stored object to pass any blocked read, we store each blocked read event in the requeue buffer so that it can be satisfied when the object for which the read is blocked arrives at the replica.

4.4.4 Event Handler/Queue

The *EventHandler* and *EventQueue* implement the event-driven task scheduling mechanism of the simulator. The EventQueue is implemented as a simple priority queue that is ordered by event arrival times. The EventHandler loops and continuously (1) retrieves the next pending event from the EventQueue, (2) extracts the ID of the server for which the event is applicable, and (3) passes the event to the OriginSimulator or ReplicaSimulator referenced by the ID. We note that the EventHandler also processes GENSTAT events by sending print commands to a statistics module (not shown in the diagram).

4.4.5 Simulator Configuration file

Although figure 4.3 does not show the configuration file used by the simulator, we describe it here for completeness. The configuration file allows setting the following parameters:

- *numFiles*: The number of files in the system.
- *algorithm*: The type of algorithm to use (e.g. Push All, TRIP, etc.)
- *replicaOriginDelayMean*: The mean network delay between a replica and the origin server.

- *randomSeed*: Random seed for the simulator.
- *averageFileSize*: The simulator uses an exponential distribution of file sizes that have a mean of this value.
- *bandwidth*: The total outgoing bandwidth budget for the origin server.
- *k*: The prefetching threshold used in the Threshold-based algorithm.

The configuration file has the simple format "<*parameter*> = <*value*>", with each such statement on a different line.

Chapter 5

Evaluation

We evaluate our traces using two approaches: by employing a trace-driven simulator and evaluating a prototype.

5.1 Simulation methodology

Our trace-driven simulator models an origin server and twenty replicas. By default we simulate a round-trip time (or 2 * *nwLatency*) of 200ms +/- 90% between the origin server and a replica.

We initially assume that the system requires (1) sequential consistency, which all of our algorithms - TRIP, Push All, Demand Only, and Threshold-based provide, and (2) a Δ-coherence guarantee of $\Delta = 60$ seconds, which Demand Only naturally meets, TRIP consciously enforces, and Push All and Treshold-based may or may not meet depending on available bandwidth (and threshold values). We will later modify these assumptions.

5.1.1 Workload

We evaluate the algorithms using a trace-based workload of the Web site of a major sporting event [1] hosted at several geographically distributed locations. The logs contain a total of 22.8 million client requests and 281 thousand writes by the origin server, and they span one day.

In order to simplify simulations we ignore certain entries in our trace file. In particular, we remove from the trace files (1) all requests that do not contain 200 or 304 as server return codes, (36.7%), (2) all dynamic requests, (13.9%), (3) entries that appear out of order in the trace files (0.58%), and (4) requests that our parser fails to parse (0.17%). We eliminate those requests with return codes other than 304 and 200 because we assume that the expensive operations at a replica are those that potentially lead to communication with the origin server. Although requests that result in error codes of 302 (server redirection) are valid requests, we remove them from our traces because those requests reappear in our trace files as requests with 304 or 200 as return codes. We remove dynamic requests because we do not have data of which underlying objects they access. We remove out-of-order requests because they pose a problem for the event queue in our trace-driven simulator. Finally, we remove requests that have valid return codes but that our trace parser fails to parse because it is conservative. Since the number of requests in the traces that are either unparseable or appear out-of-order is small, we

[1] The 2000 Summer Olympic games

do not believe that removing them significantly influences our results.

5.1.2 Prediction policy

Our interface allows a server to use any algorithm to choose the priority of an update, and this paper does not attempt to extend the state of the art in prefetch prediction. A number of standard prefetching prediction algorithms exist [24, **?**, 56, 68] or the server may make use of application-specific knowledge to prioritize an item. Our simple default heuristic for estimating the benefit/cost ratio of one update compared to another is to first approximate the probability that the new version of an object will be read before it is written as the observed read frequency of the object divided by the observed write frequency of the object and then to set the relative priority of the object to be this probability divided by the object's size [68]. This algorithm appears to be a reasonable heuristic for server push-update protocols: it favors read-often objects over write-often objects and it favors small objects over large ones.

5.2 Simulation results

Our primary simulation results are that (1) self-tuning prefetching can dramatically improve the response time of serving requests at replicas compared to demand-based strategies, (2) although a Push All strategy enjoys excellent response times by serving all requests directly from replicas' local storage, this strategy is fragile in that if update rates exceed available bandwidth for an extended period of time, the service must either violate its Δ-

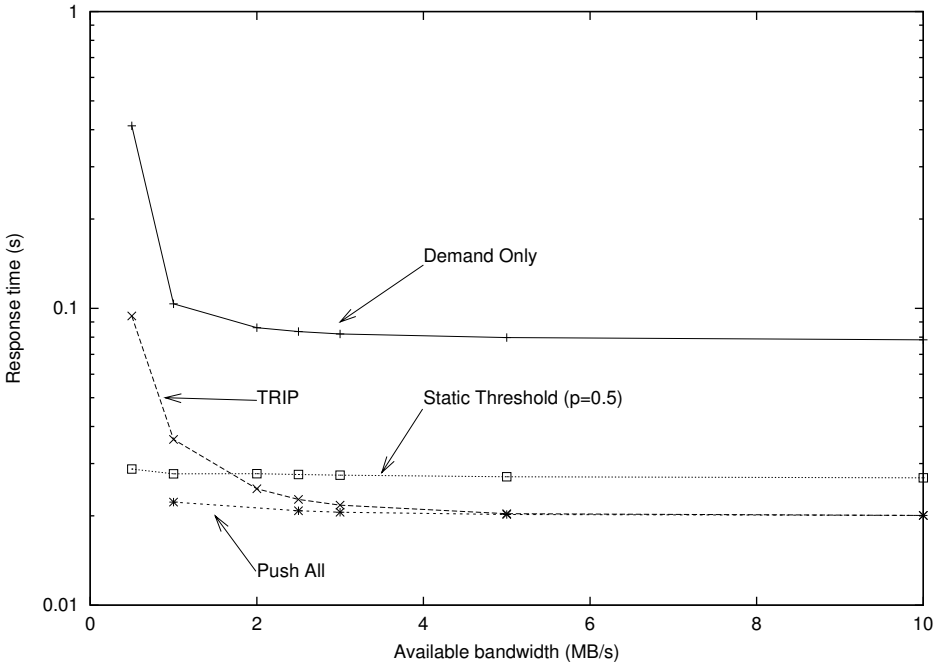

Figure 5.1: The effect of bandwidth availability on response times

consistency guarantee or become unavailable, (3) when prefetching is used, delaying application of invalidation messages by up to 60 seconds provides a modest additional improvement in response times, and (4) by maximizing the amount of valid data at replicas, prefetching can improve availability by masking disconnections between a replica and the origin server.

5.2.1 Response times and staleness

In Figure 5.1 we quantify the effects of different replication strategies on client-perceived response times as we vary available bandwidth. We assume that client requests for valid objects at the replica are satisfied in 20ms, whereas

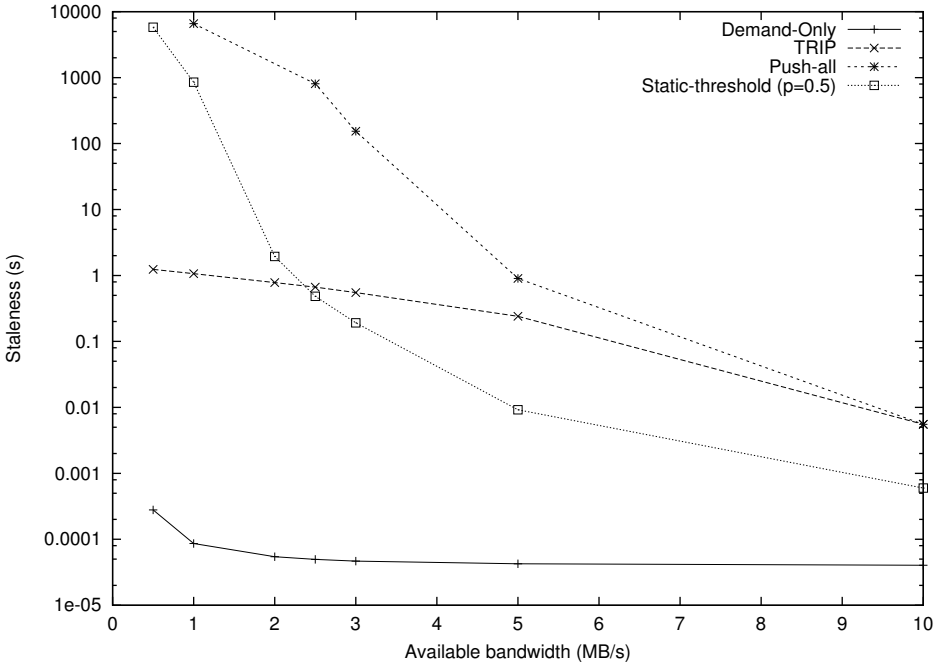

Figure 5.2: Average staleness of data served by replicas.

requests for invalidated objects are forwarded from the replica to the origin over a network with an average round-trip latency of 200ms as noted above. To put these results in perspective, Figure 5.2 plots the average *staleness* observed by a request. We define staleness as follows. If a replica serves version k of an object after the origin site has already (in simulated time) written version j ($j > k$), we define the staleness of a request to be the difference between when the request arrived at the replica and when version $k+1$ was written. To facilitate comparison across algorithms, this average staleness figure includes non-stale requests in the calculations. We omit due to space constraints a second graph that shows the (higher) average staleness observed by the subset of reads under each algorithm that receives stale data.

We also show in figures 5.1 and 5.2 the latency and staleness yielded when using the *static-threshold-prefetching* algorithm, which prefetches objects when the predicted likelihood of their being accessed exceeds a statically chosen threshold. We plot the behavior of this algorithm when it is tuned to prefetch objects that have a greater than 50% estimated chance of being accessed (denoted *Static Threhold* ($p = 0.5$) on the graph). We note that Push All and Demand Only represent extreme cases of this algorithm with thresholds of 0 (push an update regardless of its likelihood of being accessed) and 1 (only push an update if it is certain to be accessed), respectively.

The data indicate that the simple Push All algorithm provides much better response time than the Demand Only strategy, speeding up responses by a factor of at least four for all bandwidth budgets examined. However, this

comparison is a bit misleading as Figure 5.2 indicates: for bandwidth budgets below 2.1MB/s, Push All fails to deliver all of the updates and serves data that becomes increasingly stale as the simulation progresses. If the system enforces Δ-coherence with $\Delta = 60$ seconds, Push All replicas would be forced to either violate this freshness guarantee or become unavailable when the available bandwidth falls below about 5MB/s.

The static-threshold line illustrates precisely the problem with static thresholds. When the system has less than 2MB/s available bandwidth, the static-threshold algorithm yields lower response times than the TRIP algorithm. However, we note that for this bandwidth range the static-threshold algorithm also violates staleness guarantees. Similarly, when the system has more than 2MB/s bandwidth available, the static-threshold algorithm fails to utilize it to reduce response times.

The TRIP algorithm has significant advantages over Push All, Demand Only, and static-threshold. When available bandwidth exceeds 5MB/s, TRIP matches Push All's excellent response time and provides 4x and 1.3x speedups compared to the Demand Only static-threshold systems respectively. At lower bandwidths, this algorithm meets the timeliness bound of 60 seconds, but it still significantly outperforms the Demand Only strategy and yields a modest improvement over the static-threshold strategy. For example, when 2MB/s of bandwidth is available, TRIP provides speedups of 3.5 and 1.1 compared to Demand Only and static-threshold respectively, and Push All provides only an additional speedup of 1.2 despite the latter's liberties with the system's fresh-

ness requirements. Although at 2MB/s static-threshold outperforms Demand Only by a factor of 3.1, it both fails to meet our timeliness deadline and is 1.1 times slower than TRIP.

Even at low bandwidths, TRIP gets significantly better response times than the Demand Only algorithm because (a) the self-tuning network scheduler allows prefetching to occur during lulls in demand traffic even for a heavily loaded system [42] and (b) the priority queue at the origin server ensures that the prefetching that occurs is of high benefit/cost items. For example, at 500KB/s of available bandwidth, which causes significant congestion for even the Demand Only case, TRIP has more than a 3x speedup over Demand Only. TRIP's ability to exploit lulls in demand bandwidth also constitutes the reason that when the system has 2MB/s available bandwidth TRIP can outperform static-threshold while still retaining its timeliness guarantees.

5.2.2 Variations of TRIP

Figure 5.3 shows the behavior of response times under variations of TRIP. We modify the TRIP algorithm in two ways. We first vary the coherence parameter Δ on response times by setting $\Delta = 0$, which forces a replica's scheduler to apply all invalidations immediately. We then investigate the potential benefit of the *TRIP-aggressive* optimization, which sacrifices some transparency by assuming that when the application issues concurrent read requests, the requests are logically independent, which allows reads of cached objects to pass reads that have blocked (as described in Section 3.4).

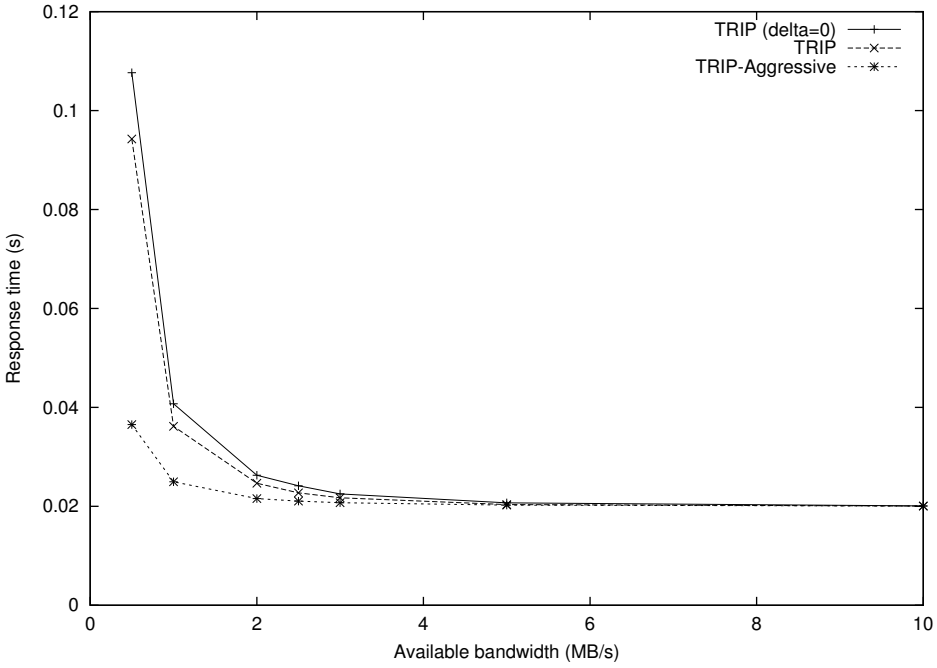

Figure 5.3: Latencies yielded by variations of TRIP

Reducing average staleness by reducing Δ below 60s inflicts a modest cost on response time, and this cost declines as available bandwidth increases. The benefit of allowing some applications to exploit independence across read requests can be substantial. For example, for a system with 500KB/s of available bandwidth, this optimization improves response time by a factor of 2.5. But, this benefit falls as available bandwidth increases, suggesting that this optimization may become less valuable as network costs fall relative to the cost of requiring programmers to carefully analyze applications to rule out the possibility of unexpected interactions [37].

5.3 Availability

We measure the replication policies' effect on availability as follows. For each of 50 runs of our simulator for a given set of parameters, we randomly choose a point in time when we assume that the origin server becomes unreachable to replicas. We simulate a failure at that moment and measure the length of time before any replica receives a request that it cannot mask due to disconnection. We refer to this duration as the *mask duration*. We assume that systems enforce Δ-coherence with $\Delta = 60$ seconds before the disconnection but that disconnected replicas maximize their mask duration by stopping their processing of invalidations and updates during disconnections and extending Δ as long as they can continue to service requests. Thus, during periods of disconnectivity, our system chooses to provide stale data rather than failing to satisfy client requests. Note that given these data, the impact of enforcing

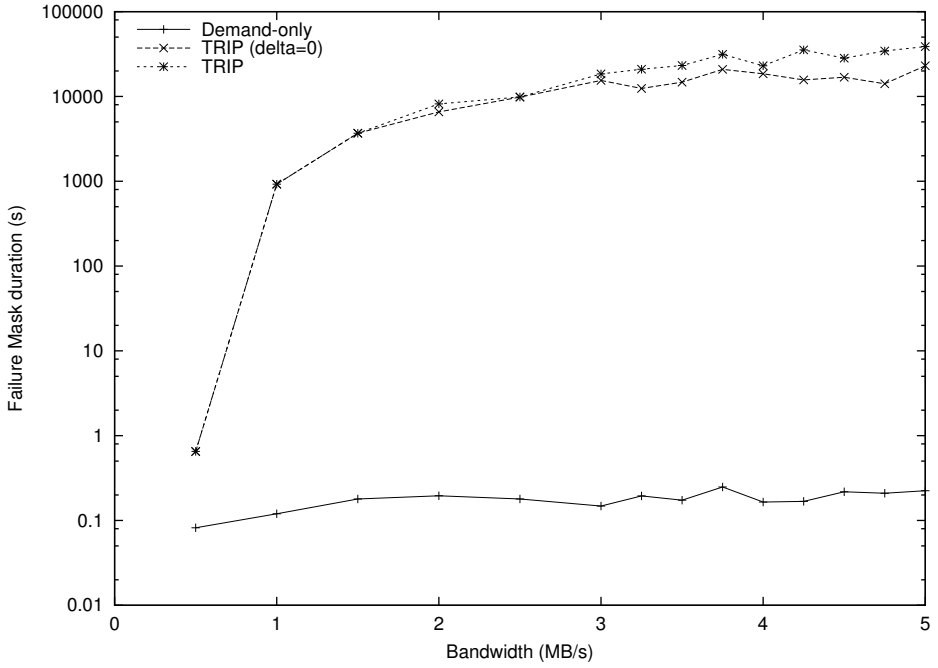

Figure 5.4: Dependence of mask duration on bandwidth.

shorter Δs during disconnections can be estimated as the minimum of the time reported here and the Δ limit enforced.

Figure 5.4 shows how the average mask duration varies with bandwidth for the TRIP, TRIP ($\Delta = 0$), and Demand Only algorithms. Because mask duration is highly sensitive to the timing of a failure, different trials show high variability. We quantify this variability in more detail in an extended technical report [54].

Note that the traditional Demand Only algorithm performs poorly. In Figure 5.4, the line closely follow $y = 0$, indicating virtually no ability to mask failures. This poor behavior arises because the first request for an object after that object is modified causes a disconnected replica to experience an unmaskable failure. On the other hand, the Push All algorithm can mask all failures due to the fact that at any point in time, the entries in a replica's cache form a sequentially consistent (though potentially stale) view of data.

The TRIP algorithm outperforms the Demand Only algorithm in the graph by maximizing the amount of local valid data. We note that both TRIP variations provide average masking times of thousands of seconds for bandwidth of 1.5MB/s and above and that providing additional bandwidth allows these systems to prefetch more data and hence mask a failure for a longer duration. As noted in Section **??**, systems may choose to relax their Δ-coherence time bound to some longer Δ' value during periods of disconnection to improve availability. These data suggest that systems may often be able to completely mask failures that last the maximum maskable duration even for

relatively large Δ' limits.

Chapter 6

Implementation

In our experience with implementing TRIP we learn that the mechanisms that we use for TRIP - (1) separation of invalidates and updates, (2) synchronizing streams of invalidates and data messages at the receiver, and (3) using self-tuning update propagation to aggresively replicate data for performance and availability - are applicable to broader scenarios.

6.1 Architecture

In this section we describe the various data structures, channels, and protocol details employed by the TRIP architecture and discuss their contribution to providing each of our requirements for transparency - consistency, availability, performance, and resource non-interference.

This section proceeds as follows. We first described a unified architecture that allows us to build both the origin server and a replica. Then, we describe how we modify the architecture and protocol to support multiple streams of invalidates and updates to allow multiple writers in the system.

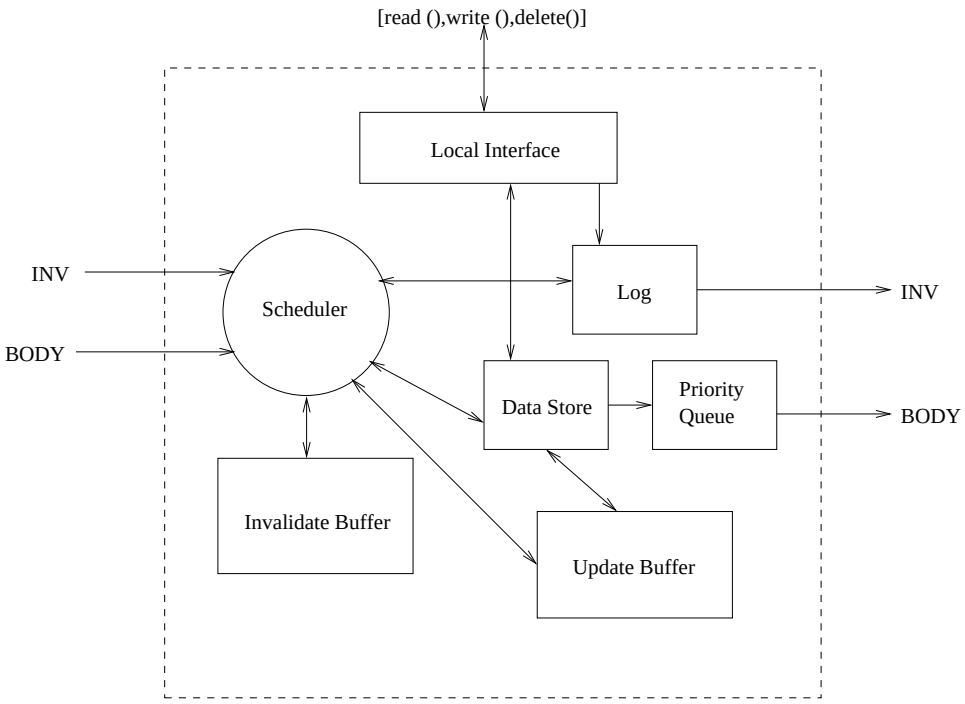

Figure 6.1: Data structures employed by TRIP

6.1.1 Data structures

Figure 6.1 provides a high-level overview of the data structures we use for TRIP. Note that because we build a set of mechanisms that can be used by both the origin server and replica, our implementation of each node provides mechanisms to both send and receive invalidates and update messages.

TRIP handles local operations - i.e. read(), write(), and delete() - using the *Log* and *DataStore* as follows. When the origin server performs a write() or delete(), our implementation (1) inserts an invalidate message into the *Log* indicating that an object is modified and (2) updates the local copy of the data in the *Data Store* with the changes. To handle a read() operation, our implementation reads the relevant object from the *Data Store* and returns it to the requesting application.

Our implementation uses three other components to apply data and metadata messages from other nodes - the $Scheduler$, $InvalidateBuffer$, and $UpdateBuffer$. We describe each of these in the context of the various mechanisms that comprise TRIP: (1) invalidation log exchange, (2) speculative pushing of updates, (3) fetching data on demand, and (4) garbage collection and checkpoint exchange.

6.1.1.1 Invalidation log exchange

We use standard log exchange protocol [64, 75] to allow the origin server to propagate invalidate messages. However, we make two key changes. First, we employ our mechanism of separating metadata from data by storing only

small invalidate messages in our log and using an alternate channel to transfer data. Second, although existing log exchange protocols synchronize updates by periodically exchanging missing log entries, our protocol allows us to treat logs as long-running streams such that newly-added entries to the local log are propagated to receivers immediately.

We use three key features of existing log exchange protocols. First, existing log exchange protocols simplify *recoverability* by allowing a replica to reconnect to the origin server after suffering from a network partition and bring itself up to date with respect to missing invalidates. Second, Finally, existing log protocols provide log *garbage collection* and *checkpoint transfer* that allows us to efficiently bound consuming resources used by the log and provide means for a heavily out-of-date replica to synchronize itself.

Our basic protocol proceeds as follows. A replica uses an outside form of communication to request the origin server to initiate exchange of invalidates. The origin server sends the replica a stream in the form <*Start time, <list of invalidates*>>, where *Start time* represents the sequence number of the first invalidate in the stream. The receiving replica compares the <*Start time*> to the value of its local clock - i.e. the largest sequence number that it has received - and rejects the stream if <*Start time*> is too large to avoid violating consistency guarantees.

The receiver *Scheduler* applies each invalidate to its $InvalidateQueue$, which stores the invalidate message until (1) its corresponding object arrives (note that it may have arrived before the invalidate), (2) it has been delayed

for more than Δ units of time, or (3) a blocked read request forces it to be accepted early. We allow a blocked read to force an invalidate message to be accepted when that read is blocked for data that is pending in the UpdateBuffer but cannot be accepted because its corresponding invalidate is queued on the InvalidateQueue behind the currently delayed invalidate. Once an invalidate message is ready to be processed, the replica applies it to the local *DataStore*.

Transparency The *Scheduler* meets our transparency goal of providing consistency by exploiting the key observation that all invalidates applied to the *DataStore* form a prefix of those sent by the origin server. Because our single-writer environment only requires providing FIFO consistency, applying invalidates in FIFO order allows the replica to provide sequential consistency. Furthermore, by delaying invalidates by up to Δ units of time the Scheduler maximizes performance and availability by increasing the likelihood that the data corresponding to an invalidate arrives before the invalidate is applied to the DataStore.

Our mechanism for transmitting invalidate messages provides transparency by meeting our goal of resource non-interference using two strategies. First, by separating data and metadata we ensure that invalidate messages are small and therefore conserve network resources compared to replication systems that push all updates to nodes [64, 75]. Second, we bound the size of the InvalidateQueue to prevent stressing memory and storage resources.

When the queue is full we pause the thread that inserts invalidates into the InvalidateQueue to pressure the sender into reducing its sending rate.

6.1.1.2 Speculative push

We use two key mechanisms to speculatively push data: The *UpdateBuffer*, and a novel *Speculative push channel*. The UpdateBuffer at the replica collects and stores data bodies to be applied by the Scheduler when the corresponding invalidate has been applied to the DataStore. The speculative push channel employs two mechanisms - a (1) PriorityQueue at the origin server that contains objects sorted in order of the benefit of preemptively pushing each to a replica, and (2) an asynchronous, unreliable, low-priority network channel that allows pushing objects using only spare bandwidth.

The mechanism that we use to provide speculative pushing provides transparency using two methods. First, the PriorityQueue and low-priority channel provide *improved availability* and *improved performance* by utilizing spare bandwidth to maximize the amount of valid data stored in a replica's DataStore.

Second, we provide resource non-interference by bounding memory and storage space using the key observations that the channel over which we speculatively push data allows either the replica or the origin server to safely discard any object. We discuss three strategies that the replica UpdateBuffer can employ when it is memory constrained:

- *Wait on full*: The *wait on full* strategy can only be implemented for the UpdateBuffer and requires it to block on an add() call until it has room to accept the new object. At the receiver, by blocking the UpdateBuffer we force the sender's network scheduler to throttle its sending rate [67] to match the receiver's ability to acccept and process update messages.

- *Drop on full*: Under the simple *Drop on full* strategy the UpdateBuffer drops all updates until it has room to add one. Our current implementation only supports this strategy to bound memory resources.

- *Priority-replace on full*: The *Priority-replace on full* strategy attempts to keep the UpdateBuffer full with the most important data by discarding low-priority objects from the buffer in favor of high-priority ones when it is full. We note that because the buffer may choose to discard multiple low-priority objects to store a single large, high-priority object, this strategy is inherently sensitive to the algorithm chosen by the user to calculate object priorities. We plan on exploring this strategy further in future work.

We note that the origin server can employ a similar strategy to *Prriority-replace on full* to maximize availability and performance at the receiving replica by keeping the PriorityQueue full with objects of highest priority. Furthermore, the origin server PriorityQueue employs an additional optimization where if an object is overwritten at the origin server it can discard the older version of that object permanently.

6.1.1.3 Demand reads

To handle requests for invalidated objects we provide replicas a provision to fetch objects from the origin server. To improve performance, the origin server prevents the replica from waiting arbitrarily long to satisfy a request by serving any demand-fetched object over a regular network priority channel that we denote the *Demand read channel*.

We note that the origin server may reply to a replica's fetch request with object data that the replica cannot apply because the corresponding invalidate message is waiting in that replica's $InvalidateBuffer$. Therefore, we implement an addional optimization where we allow the replica to stop delaying invalidate messages until each read that is blocked at that replica is satisfied. Since Demand Reads form the bottleneck to performance, by optimizing satisfying blocked reads we aid our overall goal of providing transparency.

Although we send demand replies over regular network priority, we do not consider such network consumption to be in violation with our goal of resource non-interference because we do not expend regular-priority bandwidth transferring objects that are not requested at a replica.

6.1.1.4 Garbage collection and checkpointing

We utilize two key features of existing log transfer mechanisms that we describe in further detail in [6] but outline here. First, to adhere to transparency, we allow any node to bound the size of its log of invalidate messages by periodically garbage-collecting a prefix of the log. Our implementation al-

lows the user to specify a bound on the amount of memory used to store the log.

Second, we note that since the origin server may garbage collect a prefix of its log that has not yet been seen by the replica, a replica cannot use only log transfer mechanisms for synchronization. Therefore, we utilize checkpoints [64, 75] to allow the origin server to send a snapshot of all of its state to the replica to allow the replica's log to catch up to that of the origin server.

Our checkpointing protocol differs from that described by traditional log-exchange literature in two key ways. First, because we separate data from metadata, we allow building lightweight checkoints that contain only invalidates. Second, we allow nodes to build *incremental checkpoints* that allow them to update the receiver's DataStore incrementally. In particular, when objects are organized in a hierarchical namespace (e.g. a file system [**?**]) the origin server can send a checkpoint for a high-priority subtree of the replica before sending a checkpoint for the rest of the objects. This feature allows the replica to start serving requests for data from that subtree without waiting for a complete checkpoint transfer.

Our checkpoint transfer protocol provides transparency by (1) providing network non-interference by allowing light-weight checkpoints and (2) providing performance by allowing a replica to efficiently synchronize with the origin server through the use of both incremental checkpointing and light checkpoints.

6.2 Multiple-writers

Although TRIP focuses on single-writer Information Dissemination Services, we show that our key mechanism for synchronizing streams of data and metadata can be extended to support services that involve multiple writers. We show in [6] that with the addition of more extensions to our TRIP principles - separation of data and metadata, synchronizing streams at the receiver, and utilizing safe speculative replication - can be built into a general replication framework and a novel replication toolkit.

6.2.1 Data structures

Figure 6.2 shows a high-level view of the PRACTI toolkit that illustrates our design of multi-writer TRIP. Our design is motivated by the goal of cleanly separating replication mechanisms from policies and is hence organized as two components: a *Core* that provides replication mechanisms, and a *Controller* that implements policies.

6.2.2 Challenges

We note that allowing multiple writers complicates our design space for three reasons. First, we note that because under a multi-writer environment no single node may host all data present in the system, we must allow a node to demand fetch data from any other node in the system rather than necessarily fetching it from the origin server. To support such arbitrary *data placement policies* we allow the application-provided Controller to route (1) demand-fetch

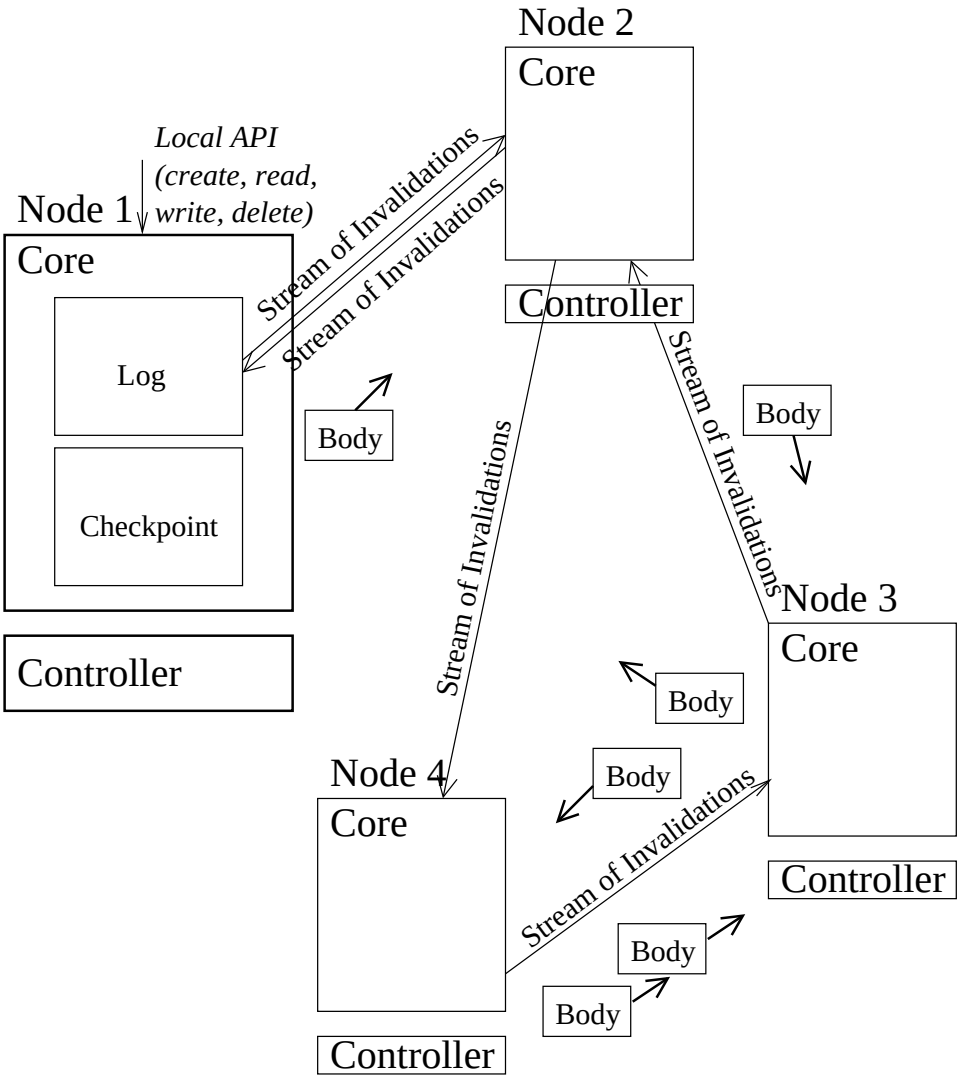

Figure 6.2: High-level view of PRACTI

requests and (2) speculative replication requests to the appropriate nodes.

Second, because any node that performs a write must be able to propagate invalidates to other nodes in the system, our architecture must allow for invalidates to originate at any node. Furthermore, we design our architecture to support *Topology Independence*, which allows invalidate messages to follow any path through the system. Our Core exploits traditional peer-to-peer log-exchange protocols [64, 75] to allow the Controller to make policy decisions regarding when a node can exchange logs with which other node.

Third, since all nodes participating in the system share the same namespace for objects, our replication system must handle conflicting writes to the same object. Our current approach to handling conflicts is limited to logging conflicts and allowing applications to view metadata information for conflicting writes [6]. We revisit the need for handling conflicts in section 6.3

6.2.3 Multi-writer design

We exploit two key benefits of our design. First, our rules to synchronize streams of invalidates and updates require only minimal modification in a multi-writer scenario - rather than using sequence number, we use pairs of $(nodeId,\ seqNo)$ to generate orderable sequence numbers that are unique across the system.

Second, our usage of existing log-exchange protocols vastly simplifies our architecture because such protocols (1) allow all nodes to perform writes and (2) allow any pair of nodes to synchronize logs to inherently support

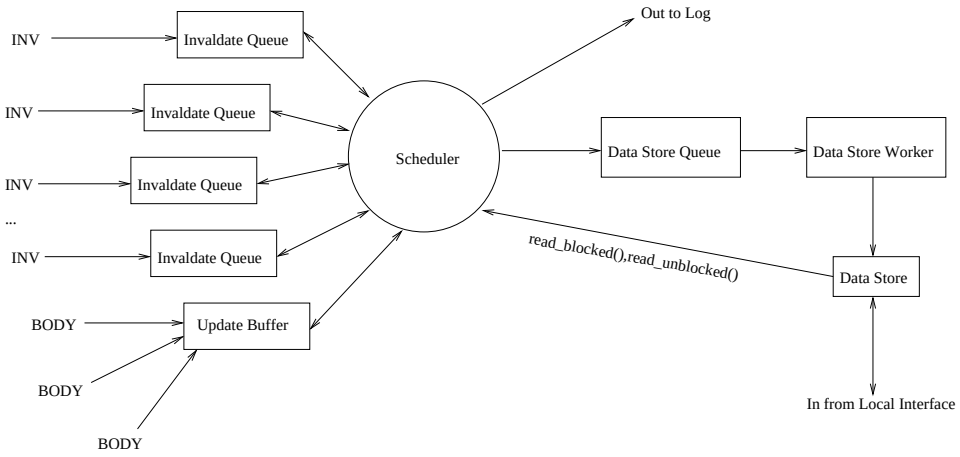

Figure 6.3: Multi-writer architecture

multiple writers and Topology Independence. A requirement of using multi-writer log-exchange protocols is that we change our description of *Start time* to include a vector clock rather than a single $seqNo$.

We note that supporting multiple writers and Topology Independence requires modification to each node's Scheduler and InvalidateQueue. We show in figure 6.3 the changes that we make to our archicture and provide details below.

6.2.4 Algorithm

Our scheduler algorithm proceeds as follows. As shown in figure 6.3, we build a separate InvalQueue for each incomming channel and assume that an outside thread reads invalidate messages and enqueues them into the appropriate channel. We also assume that an outside threads listens to *Demand*

read channels and *Speculative push* channels for update messages that it places into the single UpdateBuffer.

The multi-writer scheduler receives notification calls for 4 events - (1) the addition of an invalidate into an InvalidateQueue, (2) the addition of an object into the UpdateBuffer, (3) the blocking of a read, and (4) the unblocking of a blocked read - and reacts by placing events onto the $DataStoreQueue$ when they are ready to be processed. As shown in the figure, the $DataStoreWorker$ dequeues events from the DataStoreQueue and applies them to the DataStore.

We show the pseudo-code for the modified Scheduler in figures 3 and 4. The scheduler maintains two key data structures - a *heap* and a FIFO *dataStoreQueue*. Each entry of the heap contains tuples of (Inv, *deadline*, *processed*, $channelID$) that are sorted in increasing order of each message's *deadline*. We use a field, *processed*, to indicate whether an invalidate that is on the heap has been processed.

The scheduler also maintains a FIFO queue, the *dataStoreQueue*, that contains all outgoing messages for the $DataStoreWorker$, which repeatedly calls the $getNextMessage$ method to retrieve them and apply them to the DataStore. We note that the dataStoreQueue processes invalidate messages and update messages separately. In particular, it (1) applies invalidate messages by looking for the corresponding object in the UpdateBuffer and applying both together if possible, and (2) treats data messages as suggestions and only removes the data message from the UpdateBuffer the message can be applied.

Algorithm 3 Call to getNextMessage

```
Local state:
  dataStoreQueue;  // Messages to be applied by DataStoreWorker
  channels[];  // InvalidateQueue indexed by channelID
  heap;  // Heap of invalidates sorted by deadline

On call to getNextMessage():
  processApplicableInvalidates();
  while (dataStoreQueue.isEmpty()) do
    fixHeapTop();
    if (!heap.isEmpty()) then
      heapTopEntry = heap.peekTop();
      if ((readBlocked > 0)||(heapTopEntry.deadline > currentTime())) then
        entry = invalidateQueue[heapTopEntry.channelID].dequeue();
        dataStoreQueue.enqueue(entry.msg);
        entry.processed = true;
        while (entry! = heapTopEntry); do
          entry = invalidateQueue[heapTopEntry.channelID].dequeue();
          dataStoreQueue.enqueue(entry.msg);
          entry.processed = true;
        end while
      else
        wait(heapTopEntry.deadline = currentTime());
      end if
    else
      wait();
    end if
    processApplicableInvalidates();
  end while
  return(dataStoreQueue.dequeue());

processApplicableInvalidates():
  for (each channelID ∈ channels) do
    done = invQueue[channelID].isEmpty();
    while (!done) do
      entry = invQueue[channelID].peek();
      if ((updateBuffer.contains(entry.msg.objId, entry.msg.seqNo))||(numReadsBlocked > 0)) then
        invQueue(channelID.dequeue();
        dataStore.enqueue(entry.msg);
        entry.processed = true;
      else
        done = true;
      end if
    end while
  end for

fixHeapTop():
  done = heap.isEmpty();
  while (!done) do
    heapTopEntry = heap.peekTop();
    if (heapTopEntry.processed) then
      heap.getNext();  // Discard the top entry; processed already
      done = heap.isEmpty();  // Stop if empty
    else
      done = true;  // Found an unprocessed entry
    end if
  end while
```

Algorithm 4 Notification methods

On call to notifyAddBody(bodyMsg):
$dataStoreQueue.enqueue(bodyMsg)$;
$notifyAll()$;

On call to notifyAddInv(invQueueEntry):
$heap.add(invQueueEntry)$;
$notifyAll()$;

On call to notifyReadBlocked():
$numReadsBlocked++$;
$notifyAll()$;

On call to notifyReadUnblocked():
$numReadsBlocked--$;
$notifyAll()$;

6.3 NFS Interface

Our implementation of the NFS interface demonstrates the feasability of providing transparent replication and serves as an evaluation testbed. The NFS interface component performs bi-directional translation between NFS calls and PRACTI calls. Our interface module is built using two key NFS abstractions - an NFS *File* and NFS *Directory* - and one in-memory data structure - a *FileHandleTable*. We describe each of these below:

File Each NFS file is indexed using three identifiers: a $refObjId$ that refers to the contents of that file, a *handle* that is used by NFS methods to refer to the file, and a *name* that is exposed by the NFS client to the application:

- $refObjId$: An NFS file is stored as two PRACTI objects: (1) a *data object* that stores the file's contents and (2) an *attribute object* that stores the file's attributes. Therefore, the *refObjId* is composed of two PRACTI

object IDs: the data object ID and the attribute object ID.

- *handle*: NFS methods refer to files using platform-independent file handles. We therefore maintain a *FileHandleTable* that stores the bi-directional mapping between an NFS file handle and that file's refObjId.

- *name*: NFS exposes a file to an application as an application-specified file *name*. We maintain a mapping between a file name and its refObjId in our *Directory* abstraction.

Directory We represent an NFS directory as a table that contains a single entry for each file that is logically contained in that directory according to the namespace visible to the NFS client. Each entry contains (1) a file *name* that is used as an index into the table and (2) the $refObjId$ for that file.

To maintain NFS semantics, we note that an NFS directory is also itself a file. Therefore, (1) a directory can be contained in another directory, and (2) modifications to a directory (or its associated attributes) are propagated to all replicas.

Directories map file names to $refObjIds$ rather than directly to file handles for two reasons. First, we treat file handles to be *ephemeral*; i.e., they are likely to change over time and after a node restarts. Second, file handles are locally generated and vary across nodes. The $refObjId$ of a file, on the other hand, serves as an immutable pointer to that file.

FileHandleTable The NFS protocol uses file handles rather than file names for three key reasons. First, file handles allow for more *efficient server implementation* because they allow servers to encode file location information (e.g. the i-node number where the file is located on disk) into the file handle such that to serve future requests the server would be able to efficiently retrieve a file based on its file handle. Second, file handles allow for *cross file system compatibility*, because they allow the file name that is exposed to the client application to vary across clients and applications. For example, a client can use either / or \ as its path separator transparent to the NFS server. Finally, the usage of handles allows for better *parallel access* to files because NFS handles do not change through client calls to rename(). Thus, one client can change the name of a file while another can continue accessing it.

Therefore, to adhere to NFS semantics, we build a $FileHandleTable$ that maintains a list of $(handle, refObjId)$ pairs. Because some NFS calls supply an NFS handle whereas other request one, our table is indexed by both the NFS file handle and by the refObjId.

We utilize the fact that NFS handles are ephemral to gain three benefits. First, because an NFS client cannot expect handles to be long-lived, we bound the size of our FileHandleTable by discarding old, unused handles (e.g. using an LRU replacement policy). Second, we efficiency of access by storing our table only in memory because it can be discarded on reboot. Finally, we generate handles for files only as they are requested and thus do not spend resources generating and storing handles for unaccessed files.

6.3.1 NFS consistency

Although NFS provides only loose requirements on the consistency of the files that are exposed to users, NFS requires that each operation be *atomic* - i.e., the system should never be visible in a state where a particular operation has only partly completed. For example, the successful completion of a write to a file should (1) update file attributes to reflect the time of the new write, and (2) update the file data.

We support such operations through the use of *atomic writes*. Atomic writes are a series of writes that are applied by each replica together. In our PRACTI implementation, we implement atomic writes through the use of *atomic invalidate* messages that form metadata that contain the names of multiple objects, flags indicating which object is deleted and which is written, and a single timestamp for the operation.

6.3.2 Operations

Algorithm 5 State required by described methods

State

practi // Implementation of PRACTI
fhTable // Instance of FileHandleTable
lockManager // Provides reader/writer locks

Algorithms 6-10 describe five key operations that our NFS interface implementation must support: *create*, *rename*, *lookup*, *read*, and *write*. Figure 5 describes the three components that we assume present in the system: an instance of PRACTI, an instance of the $FileHandleTable$, and *lock manager*

Algorithm 6 Create() method

create(dirFileHandle, name, newAttributes):

$dirRefObjId = fhTable.get(dirFileHandle)$
$fileRefObjId = makeNewRefObjId(dirRefObjId, name)$
$dirLock = lockManager.acquireWriteLock(dirRefObjId)$
$fileLock = lockManager.acquireWriteLock(fileRefObjId)$
$dir = readDir(dirRefObjId)$
$dir.add(name, fileRefObjId)$
$op_1 = write\ dir\ to\ dirRefObjId.dataObjId$
$op_2 = write\ dir.attributes\ to\ dirRefObjId.attrObjId$
$op_3 = create\ fileRefObjId.dataObjId$
$op_4 = write\ newAttributes\ to\ fileRefObjId.attrObjId$
$practi.write(op)$
$lockManager.release(fileLock)$
$lockManager.reelase(dirLock)$

Algorithm 7 Rename() method

rename(fromDirFH, name, toDirFH, newName):

if $(fromDirFH \neq toDirFH)$ **then**
  $fromDirRefObjId = fhTable.get(fromDirFH)$
  $toDirRefObjId = fhTable.get(toDirFH)$
  $fromDirLock = lockManager.acquireWriteLock(fromDirRefObjId)$
  $toDirLock = lockManager.acquireWriteLock(toDirRefObjId)$
  $refObjId = fromDir.get(name)$
  $fromDir = readDir(fromDirRefObjId)$
  $toDir = readDir(toDirRefObjId)$
  $toDir.add(newName, refObjId)$
  $fromDir.remove(name)$
  $op_1 = write\ fromDir\ to\ fromDirRefObjId.dataObjId$
  $op_2 = write\ fromDir.attributes\ to\ fromDirRefObjId.attrObjId$
  $op_3 = write\ toDir\ to\ toDirRefObjId.dataObjId$
  $op_4 = write\ toDir.attributes\ to\ toDirRefObjId.attrObjId$
  $practi.write(op)$
  $lockManager.release(toDirLock)$
  $lockManager.release(fromDirLock)$
else
  $dirRefObjId = fhTable.get(fromDirFH)$
  $dirLock = lockManager.acquireWriteLock(dirRefObjId)$
  $refObjId = fromDir.get(name)$
  $dir = readDir(dirRefObjId)$
  $dir.remove(name)$
  $op_1 = write\ dir\ to\ refObjId.dataObjId$
  $op_2 = write\ dir.attributes\ to\ refObjId.attrObjId$
  $practi.write(op)$
  $dir.add(newName, refObjId)$
  $lockManager.release(dirLock)$
end if

Algorithm 8 Lookup() method

lookup(dirFileHandle, name):

$dirRefObjId = fhTable.get(dirFileHandle)$
$dirLock = lockManager.acquireReadLock(dirRefObjId)$
$dir = readDir(dirRefObjId)$
$fileRefObjId = dir.getRefObjId(name)$
$handle = fhCache.getHandle(fileRefObjId)$ // $Create\ if\ required$
$lockManager.release(dirLock)$
$return(handle)$

Algorithm 9 Read() method

read(handle, offset, length):

$refObjId = fhTable.get(handle)$
$lock = lockManager.acquireReadLock(refObjId)$
$data = practi.read(refObjId.dataObjId, offset, length)$
$lockManager.release(lock)$
$return(data)$

Algorithm 10 Write() method

write(handle, offset, data):

$refObjId = fhTable.get(handle)$
$lock = lockManager.acquireWriteLock(refObjId)$
$op_1 = write\ data\ to\ refObjId.dataObjId\ at\ offset$
$op_2 = write\ new\ attributes\ to\ refObjId.attrObjId$
$practi.write(op)$
$lockManager.release(lock)$

that is responsible for maintaining synchronization between threads by allowing threads to acquire/release shared and exclusive (reader and writer) locks for files.

Our algorithms call a method, $makeNewRefObjId$, that creates an object ID from the object ID of the parent directory and the file name. We note that to support successive calls to create() and rename(), our algorithm must ensure that each call creates a unique object ID. Therefore, our current implementation creates a new ObjId by concatenating (1) the object ID of the parent directory, (2) the new file name, and (3) an *instance number* that is incremented each time we encounter the same directory object ID and file name.

6.3.3 Limitations

The design of our NFS interface faces five key limitations.

First, our system does not currently support hard links, although we speculate that it is possible to support hard links by storing with file attributes a reference count that is incremented when a new link to the file is created and decremented when a link is deleted. When the reference count drops to 0, we discard the file and free all associated storage resources.

Second, our system does not enforce any type of security. In particular, we allow any user to access any file without authenticating the user.

Third, we trade-off efficiency for portability and simplicity. In particular, we gain portability by implementing our NFS interface component as

a user-level NFS loopback server?? written in Java. We gain simplicity by implementing our NFS Directory structure to write the entire directory after each modification, and by liberally using possibly unnecessary memory copies. We speculate that our NFS interface component could be tuned significantly for performance.

Fourth, our implementation does not conveniently support partial replication because the namespace imposed on data that is exposed to the application may not mirror the namespace of objects stored in PRACTI. For example, due to a call to the rename() operation, a file that was originally created under "/a/" may be moved to "/b/", although the PRACTI object ID for that object would continue to place the object under "/a/". Thus, any node that chooses to partially replicate the subtree under "/a/" would receive that object - although it has been logically moved to another directory - and any node that chooses to partially replicate the subtree under "/b/" would not automatically receive updates to the object. Furthermore, the larger the number of times that files are renamed, the larger the discrepancy between the PRACTI object ID namespace and the NFS namespace, leading to potentially large amounts of wasted bandwidth trasferring objects that are not required at the receiver.

Conflict Detection Finally, we face the significant limitation that we do not currently support conflict detection. In particular, parallel operations that affect the same files may cause all but one of those operations to be discarded. For example, if a node adds a file to directory while another node adds a

separate file to the same directory, the system will choose one of the add operations as being the last write operation and will thus discard the other.

Although our design of the interface to handling conflict resolution is work in progress, we provide a brief discussion of our approach below. We focus our discussion on conflict detection for the NFS interface module and identify three key conflict patterns that arise in various situations:

1. Simultaneous additions/removals of different files in a directory: Since conflicting additions and removals of different files are independent operations, we perform them all to the same directory safely.

2. Simultaneous addition/removal of the same file in a directory: To avoid loss of data, we choose the conservative approach of letting an add operation supersede a remove operation. Thus, a file is only removed from a directory if all conflicting writes for a file to a given directory delete that file. We also ensure that all additions/removals for a single file get merged into a single operation; e.g. we ensure that we do not add a file twice to a directory and create two entries for the same file.

3. Simultaneously delete a file and modify it: We also handle this case by conservatively choosing to allow the file modification to supersede its deletion; therefore, we do not remove the file from the directory and discard the delete request.

Although the first two classes of conflicts can be detected as conflicting writes to the same diretory; the third appears as conflicting writes to the

same file. We note that each NFS operation that modifies data gets translated into a single atomic invalidate; therefore, our conflict detection module must be careful to isolate the actual conflicting writes when examining a pair of conflicting atomic invalidates.

We note that currently PRACTI provides conflict detection support by detecting conflicts when merging logs of metadata and then appending entries to a *conflict log*. However, our conflict handling mechanism outlined above requires that we examine the actual content of conflicting writes to directories to resolve the conflict, and due to our separation of data and metadata paths such content may not be available at a node. Therefore we anticipate that in the future we could build more support for conflict resolution by allowing a node to request particular versions of objects to resolve conflicts between them.

6.3.4 Alternate implementations

We discuss two possible implementations that differ in their ability to support partial replication.

First, we consider the option of building object IDs by allowing a node to directly build a refObjId to refer to the file and have that be the handle. Such an approach would have two benefits. First, a node would never have to purge a handle. Second, we could avoid one lookup and directly pass the handle to the underlying PRACTI system. However, such an approach has the key disadvantage that the object ID of a file would have no correlation

with the file name. Therefore, we would not be able to benefit from the fact that partially replicating a subtree of the file system namespace allows for very compact representation of the set of replicated data. We could remove the existence of the FileHandleTable.

Second, we consider the option of maintaining complete similarity between the objId of an object and its name in the NFS namespace. The key advantage of this approach is that it enables intuitive partial replication, because any user that chooses to replicate a subtree of the NFS namespace would be able to concisely describe the same subtree in the PRACTI object ID namespace. However, implementing such an operation would be costly; in particular, we must either (1) add an explicit rename() method to the PRACTI interface or (2) implement the NFS version of the rename() method to atomically copy the source file to the target file name, delete the original, and then change the handle mapping in the FileHandleTable. Whereas the first approach requires significant changes to our PRACTI implementation, the second approach yields a very expensive implementation of the rename() method because each call copies all data present in a file.

Chapter 7

Related work

Since TRIP is the first system to provide *transparent* replication, existing and proposed replication systems do not provide the same consistency, availability, performance or resource non-interference properties. We broadly classify existing replication systems based on their choice of consistency and resource consumption behaviors and explore the trade-offs they make to provide perfomance and availability.

7.1 Consistency

Most existing replication systems do not provide sequential consistency with tunable staleness and hence do not meet our goals of transparent replication. In particular, most proposed Internet-scale data replication systems focus on ensuring various levels of coherence or staleness or both [20, 44, 50, 53, 70, 72, 71], but few provide explicit consistency guarantees. Unfortunately, Frigo notes that even strong coherence is considerably weaker than sequential consistency [29]. Bradley and Bestavros [10] argue that increasingly complex Internet-scale services will demand sequential consistency and propose a vector-clock-based algorithm for achieving it. They focus on de-

veloping a backwards-compatible browser-server protocol and do not explore prefetching. The IBM Sporting and Event CDN system uses a push-all replication strategy and enforces delta coherence via invalidations [15]. Akamai's EdgeSuite [3] primarily relies on demand reads and enforces delta coherence via polling with stronger consistency available via object renaming. Burns et al. [12] discuss a *publish consistency* model of consistency that is useful for web workloads and show that the normal multi-writer sequential consistency provided by file systems may be prohibitively expensive for use by web services.

Several replicated database systems have explored ways to allow different updates to specify different consistency requirements. Lazy Replication [46] allows an update to enforce causal, sequential, or linearizable consistency. Bayou [57] maintains causal consistency at all times and asynchronously reorders operations to eventually reach a global sequentially-consistent ordering of updates that may differ considerably from the causal order seen when a node first applies a given update. These systems both focus on multi-writer environments and eventually propagate all updates to all replicas. Yu and Vahdat [74] show that in such systems minimizing the time between when an update occurs and when it propagates maximizes system availability for any given consistency constraint. Our protocol exploits this observation for dissemination workloads by integrating consistency and self-tuning prefetch.

Several distributed object systems have proposed using different consistency semantics and implementations customized for different objects. Globe [66] uses the notion of distributed objects to allow application developers to

exploit application-specific consistency semantics to replicate objects. Gao et al. [31] follow a similar strategy replicate objects to support edge servers running an e-commerce TPC-W benchmark. Our study focuses on a more restricted single-write update model, but it provides sequential consistency guarantees, supports generic data, and integrates self-tuning update propagation. These object-based approaches would allow our system to be incorporated as a building block to improve the performance of dissemination objects within their systems.

Our argument for providing sequential consistency is similar in spirit to Hill's position that multiprocessors should support simple memory consistency models like sequential consistency rather than weaker models [37]. Hill argues that speculative execution reduces the performance benefit that weaker models provide to the point that their additional complexity is not worth it. We similarly argue that for dissemination workloads, as technology trends reduce the cost of bandwidth, prefetching can reduce the cost of sequential consistency so that little additional benefit is gained by using a weaker model and exposing more complexity to the programmer.

7.2 Network resource consumption

Most replication systems violate our goal of transparency because they use static replication policies such as always-conservative demand fetching [3, 18, 38, 41] which fails to utilize speculative replication to provide performance and availability, or always-aggressive push-all [14, 57, 74], which violates our

desired property of resource non-interference.

The literature discusses several algorithms that employ hand-tuned threshold-based prefetching [24, 35, 36, 56, 1, 7]. For example, Acharya et al. [1] provide mechanisms to disseminate data in one-to-many scenarios on broadcast media but require the user to specify a bound on the amount of bandwidth to be used for speculative replication. The authors of [7, 24] propose static prefetching algorithms where replicas search for temporally correlated requests and use Markov statistical models to infer the benefit of prefetching a given object when a request for another object arrives. However, these algorithms also require static thresholds to throttle the rate of prefetching data and hence may violate our transparency requirements.

Davison et al. [23] propose using a connectionless transport protocol and using low priority datagrams (the infrastructure for which is assumed) to reduce network interference. However, their work requires clients to specify the bandwidth dedicated to prefetch traffic.

Chen et al. [19] study content delivery and caching in publish/subscribe systems and discuss methods to estimate the benefit of caching pages. We speculate that their mechanisms to computing prefetch benefits of pages can be extended for use in TRIP.

Crovella et al. [21] show that a window-based rate controlling strategy for sending prefetched data leads to less bursty traffic and smaller network queue lengths. In work done by other researchers at our University, we describe

a threshold-free prefetching system called NPS [42] that like TRIP makes use of TCP-Nice [67] to avoid network interference. The rest of NPS's design is quite different than TRIP's: NPS focuses on supporting prefetching of soon-to-be-accessed objects by client browsers rather than pushing of updates by origin servers to replicas, and it does not consider the problem of maintaining consistency for data that may be prefetched long before being used.

Fei [25] simulates a threshold policy that uses an object's read rate, its write rate, and the network topology to choose between multicasting updates on one hand or multicasting invalidates and unicasting demand read replies. SPREAD [60] dynamically builds application-level invalidation and multicast hierarchies for each volume of objects, with each proxy cache using a threshold scheme to choose between polling, joining the invalidation tree, and joining the update tree for each volume. Li and Cheriton [50] propose a push-all multicast strategy that separates data into volumes so that a replica only receives updates for volumes it has referenced. Bullet [43] describes an algorithm to multicast data over a mesh rather than a tree. All of these multicast proposals provide best-effort semantics by immediately applying all messages to reduce the risk that reordering compromises consistency and to minimize real-time staleness.

Franklin et al. [28] characterize data delivery algorithms along three axes: *push v. pull*, *periodic v. aperiodic*, and *unicast v. 1-to-N*, and show that most data delivery mechanisms can be described using their classification scheme. In our study, Push All is an example of a unicast-aperiodic-push

scheme, Demand Only is an example of a unicast-aperiodic-pull scheme, and the TRIP algorithm tunes itself to available bandwidth and spans the range between the two schemes. Our paper considers only the unicast, aperiodic class of algorithms.

We note that CODA [41] provides mechanisms for both speculative replication (hoarding) and invalidations. However, CODA requires the user to choose what data to speculatively replicate and does not use background data to do so. Furthermore, although CODA is designed for mobile clients, it does require global synchronization across all connected clients that replicate an object when that object is modified.

Chapter 8

Conclusions

We discuss in this dissertation an approach to *transparent* replication for *Information Dissemination Services*, a small but important class of services for which we meet our goal of transparency to a level acceptable to such services. This dissertation make two key contributions.

First, we describe *TRIP* for which we (1) develop a novel dual-channel replication algorithm that utilizes spare network background traffic to speculatively replicate data in a safe, non-interfering fashion, (2) show how to integrate safe speculative replication with mechanisms that use invalidates to provide consistency, and (3) demonstrate how our combination of consistency and safe speculative replication allows us to provide near-ideal consistency, performance, and availability for Information Dissemination Services.

Second, we show that the core principles behind building TRIP can be extended to build a new replication framework and more general replication toolkit. In particular, we show that it is possible to extend our dual-queue mechanisms developed for TRIP to a multi-writer environment where nodes can synchronize multiple incoming streams of data and consistency information. Our extension allows providing consistency for arbitrary topologies - two

key properties provided by the PRACTI [6] architecture.

Appendix

In this chapter we prove that the following condition provides sequential consistency:

C1 A replica must apply all invalidates with sequence numbers less than N to its storage before it can apply an invalidation, update, or demand reply with sequence number N.

Theorem 1 *Condition C1 provides sequential consistency*

Proof Our proof proceeds as follows. We note that sequential consistency imposes the following constraint on a replication system:

"The result of any execution is the same as if the [read and write] operations by all processes were executed in some sequential order and the operations of each individual processor appear in this sequence in the order specified by its program." [48]

Therefore, our approach consists of constructing such a sequence, s, and proving the following two lemmas:

- **Lemma**: Sequence s is a totally ordered sequence

- **Lemma**: The result of running an application on TRIP is the same as if each operation were executed in the order specified by s and events from an individual node appear in s in the order specified by the application.

We describe the construction of sequence s in definition 1 below:

Definition 1 *We define a schedule s as an ordered sequence of all TRIP operations that occur during an execution.*

Construction: We construct s as follows. We assign to each TRIP operation q a 3-tuple timestamp ($write\_stamp$, $node\_ID$, $read\_stamp$) where we define each field as follows:

1. $node\_ID$: The ID of the node where the event occurred. We set the $node\_ID$ for the origin server to 0 - hence all writes have this field set to 0 - and assume that each replica has a unique $node\_ID$ that is larger than 0.

2. $write\_stamp$: The sequence number of the most recent invalidate message processed at the node that performs the operation.

3. $read\_stamp$: A local logical clock at each node that increases with each read performed by that node.

Each timestamp is thus sorted by (1) $write\_stamps$ such that an event with a smaller write stamp appears earlier in the schedule, (2) $node\_ID$ such

that ties between events with equal write stamps are broken using the node ID, and (3) *read_stamp* such that two operations that have the same *write_stamp* and occur on the same node get ordered by the *read_stamp*.

Lemma 1.1 *Schedule s is a totally ordered sequence*

Proof The proof of lemma 1.1 follows from the simple observation that (1) all writes have different *write_num* values, (2) each read at a given node has a unique *read_num* value, and (3) ties between reads across replicas are broken using the $node\_ID$ of each replica.

■

Lemma 1.2 *The result of running an application on TRIP is the same as if each operation were executed in the order specified by s and events from an individual node appear in s in the order specified by the application.*

Proof In order to show that the results of the sequential execution of schedule s match those of running an application on TRIP, we note that schedule s must meet four requirements: every read that reads version k of an object is ordered (1) before the write that modifies version k, (2) after the write that results in the generation of version k, and (3) after any read that happened previously on the same node. Furthermore, we note that write requests must appear in s in the order in which they occur at the origin.

We break our proof of lemma 1.2 into two parts: (1) lemma 1.3, which shows that two operations performed on the same node appear in the correct

order in s, and (2) lemma 1.4 shows that operations performed on different nodes appear in the correct order in s. We use the notation $\rightarrow$ to indicate a dependence; thus, $q1 \rightarrow q2$ indicates that operation q1 must appear in schedule s before operation $q2$. Furthermore, we utilize the notation $timestamp[q]$ to indicate the 3-tuple timestamp for operation q and use . to specify a particular field of the operation timestamp. Thus, $timestamp[q1].read\_num$ refers to the value of the read_num logical clock at the node where q1 occurs.

Lemma 1.3 *For requests* $q1$ *and* $q2$ *where* $timestamp[q1].node\_ID = timestamp[q2].node\_ID$*, if* $q1 \rightarrow q2$ *then* $q1$ *appears before* $q2$ *in* s*.*

Proof If the node that satisfies $q1$ and $q2$ is the origin, $timestamp[q2].write\_num$ must be larger than $timestamp[q1].write\_num$ because each write has a unique, increasing $write\_num$.

If the operations arrive at a replica, we note that $timestamp[q2].write\_num \leq timestamp[q1].write\_num$ must be true. If $timestamp[q1].write\_num = timestamp[q2].write\_num$ then $timestamp[q2].read\_num$ must be larger than $timestamp[q1].read\_num$ because $read\_num$ increases at each replica when it performs a read operation.

Lemma 1.4 *For requests* $q1$ *and* $q2$ *where* $timestamp[q1].node\_ID \neq timestamp[q2].node\_ID$*, if* $q1 \rightarrow q2$ *then* $q1$ *appears before* $q2$ *in the schedule.*

Proof We note that since $q1$ and $q2$ have a causal relationship, there must exist (1) some object o, (2) a read request r that reads o and (3) a write request w that modifies o such that one of the following conditions is true:

Case 1:

- $q1 \rightarrow r$ or $q1 = r$
- $w \rightarrow q2$ or $w = q2$
- $r \rightarrow w$

We note that since r was satisfied with a version of object o that is older than or equal to the one written by w, the replica that satisfied request r satisfied it before it processed the invalidate for write w. Thus, $timestamp[r].write\_num$ is necessarily smaller than $timstamp[w].write\_num$.

Case 2:

- $q1 \rightarrow w$ or $q1 = w$
- $r \rightarrow q2$ or $r = q2$
- $w \rightarrow r$

We note that if r reads a version of o that is written by a write message after w, $timestamp[r].write\_num$ is necessarily smaller than $timestamp[w].write\_num$. If r reads the same version of o that is written

by w, then although $timestamp[r].write\_num = timestamp[w].write\_num$, $timestamp[r].node\_ID$ is necessarily larger than $timestamp[w].node\_ID$ since $timestamp[w].node\_ID = 0$.

■

Theorem 1 *Condition C*1 *provides sequential consistency*

Proof The proof of theorem 1 (revisited above) follows from the definition of sequential consistency [48] and lemma 1.1 and 1.2.

■

Bibliography

[1] Swarup Acharya, Michael Franklin, and Stanley Zdonik. Balancing push and pull for data broadcast. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 183–194. ACM Press, 1997.

[2] S. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[3] Turbo-charging dynamic web data with akamai edgesuite. Akamai White Paper, 2001.

[4] A. Awadallah and M. Rosenblum. The vMatrix: A network of virtual machine monitors for dynamic content distribution. In *Internat. Workshop on Web Caching and Content Distribution*, August 2002.

[5] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-Volatile Memory for Fast, Reliable File Systems. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 10–22, September 1992.

[6] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. Practi replication. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2006.

[7] A. Bestavros. Speculative Data Dissemination and Service to Reduce Server Load, Network Traffic, and Service Time in Distributed Information Systems. In *International Conference on Data Engineering*, March 1996.

[8] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. Technical Report RFC 2475, IETF, December 1998.

[9] R. Braden, D. Clark, and S. Shenker. Integrated services in the internet architecture: an overview. Technical Report RFC 1633, IETF, June 1994.

[10] A. Bradley and A. Bestavros. Basis token consistency: Supporting strong web cache consistency. In *GLOBECOMM*, 2003.

[11] E. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, July/August 2001.

[12] R. Burns, R. Rees, and D. Long. Consistency and locking for distributing updates to web servers using a file system. In *Workshop on Performance and Architecture of Web Servers*, June 2000.

[13] P. Cao, J. Zhang, and Kevin Beach. Active Cache: Caching Dynamic Contents on the Web. In *Proc. Middleware 98*, 1998.

[14] Jim Challenger, Paul Dantzig, and Arun Iyengar. A scalable and highly available system for serving dynamic data at frequently accessed web sites. In *ACM/IEEE, Supercomputing '98*, November 1998.

[15] Jim Challenger, Paul Dantzig, Arun Iyengar, Mark Squillante, and Li Zhang. Efficiently serving dynamic data at highly accessed web sites. *IEEE/ACM Transactions on Networking*, 12(2):233–246, April 2004.

[16] S. Chandra, M. Dahlin, B. Richards, R. Wang, T. Anderson, and J. Larus. Experience with a Language for Writing Coherence Protocols. In *USENIX Conference on Domain-Specific Languages*, October 1997.

[17] S. Chandra, B. Richards, and J. Larus. Teapot: Language Support for Writing Memory Coherence Protocols. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, May 1996.

[18] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. A Hierarchical Internet Object Cache. In *1996 USENIX Technical Conference*, January 1996.

[19] Mao Chen, Andrea LaPaugh, and Jaswinder Pal Singh. Content distribution for publish/subscribe services. In *Proceedings of the International Middleware Conference*, 2003.

[20] E. Cohen, B. Krishnamurthy, and J. Rexford. Improving End-to-End Performance of the Web Using Server Volumes and Proxy Filters. In *ACM SIGCOMM Conference*, 1998.

[21] M. Crovella and P. Barford. The network effects of prefetching. In *Proc. of IEEE Infocom*, 1998.

[22] DARPA. Rfc 791 - internet protocol, Sep 1981.

[23] Brian D. Davison and Vincenzo Liberatore. Pushing politely: Improving Web responsiveness one packet at a time (extended abstract). *Performance Evaluation Review*, 28(2):43–49, September 2000.

[24] D. Duchamp. Prefetching Hyperlinks. In *2nd USENIX Symposium on Internet Technologies and Systems*, October 1999.

[25] Zongming Fei. A novel approach to managing consistency in content distribution networks. In *Internat. Workshop on Web Caching and Content Distribution*, June 2001.

[26] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Misinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, IETF, June 1999.

[27] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. In *Global Grid Forum*, June 2002.

[28] Michael Franklin and Stanley Zdonik. A framework for scalable dissemination-based systems. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 94–105. ACM Press, 1997.

[29] M. Frigo. The weakest reasonable memory. Master's thesis, MIT, 1998.

[30] M. Frigo and V. Luchangco. Computation-Centric Memory Models. In *Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1998.

[31] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar. Application specific data replication for edge services. In *International World Wide Web Conference*, May 2003.

[32] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM Press.

[33] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[34] S. Gribble, E. Brewer, J. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction, 2000.

[35] J. Griffioen and R. Appleton. Automatic Prefetching in a WAN. In *IEEE Workshop on Advances in Parallel and Distributed Systems*, October 1993.

[36] J. Gwertzman and M. Seltzer. The case for geographical pushcaching. In *HOTOS95*, pages 51–55, May 1995.

[37] M. Hill. Multiprocessors should support simple memory consistency models,. In *IEEE Computer*, August 1998.

[38] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Trans. on Computer Systems*, 6(1):51–81, February 1988.

[39] MQSeries: An introduction to messaging and queueing. IBM Corporation GC33-0805-01, July 1995.

[40] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Twenty-ninth ACM Symposium on Theory of Computing*, 1997.

[41] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Trans. on Computer Systems*, 10(1):3–25, February 1992.

[42] R. Kokku, P. Yalagandula, A. Venkatramani, and M. Dahlin. Nps: A non-interfering deployable web prefetching system. In *4th USENIX Symposium on Internet Technologies and Systems*, March 2003.

[43] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: high bandwidth data dissemination using an overlay mesh. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operat-*

ing systems principles, pages 282–297, New York, NY, USA, 2003. ACM Press.

[44] B. Krishnamurthy and C. Wills. Piggyback Server Invalidation for Proxy Cache Coherency. In *Proc. of the Seventh International World Wide Web Conference*, 1998.

[45] A. Kuzmanovic and E. Knightly. Tcp-lp: A distributed algorithm for low priority data transfer, 2003.

[46] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Trans. on Computer Systems*, 10(4):360–361, November 1992.

[47] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7), July 1978.

[48] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[49] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[50] D. Li and D. R. Cheriton. Scalable web caching of frequently updated objects using reliable multicast. In *Proceedings of the 1999 Usenix Symposium on Internet Technologies and Systems (USITS'99)*, October 1999.

[51] R. Lipton and J. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton, 1988.

[52] Sun Microsystems. Rfc 1094 - nfs: Network file system protocol specification, Mar 1989.

[53] M. Mikhailov and C. Wills. Evaluating a new approach to strong web cache consistency with snapshots of collected content. In *International World Wide Web Conference*, May 2003.

[54] A. Nayate, M. Dahlin, and A. Iyengar. Integrating Prefetching and Invalidation for Transparent Replication of Dissemination Services. Technical Report TR-03-44, University of Texas at Austin, December 2003.

[55] K. Nichols, V. Jacobson, and L. Zhang. A two-bit differentiated services architecture for the internet. Technical Report RFC 2638, IETF, July 1999.

[56] V. Padmanabhan and J. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. In *ACM SIGCOMM Conference*, pages 22–36, July 1996.

[57] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *16th ACM Symposium on Operating Systems Principles*, October 1997.

[58] C. Plaxton, R. Rajaram, and A. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Ninth Annual ACM Sym-*

posium on Parallel Algorithms and Architectures, pages 311–320, June 1997.

[59] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *ACM SIGCOMM Conference*, pages 161–172, New York, NY, USA, 2001. ACM Press.

[60] P. Rodriguez and S. Sibal. SPREAD: Scalable platform for reliable and efficient automated distribution. In *Proceedings of the 9th International WWW Conference*, May 2000.

[61] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the pangaea wide-area file system. *SIGOPS Oper. Syst. Rev.*, 36(SI):15–30, 2002.

[62] A. Siegel. *Performance in Flexible Distributed File Systems*. PhD thesis, Cornell, 1992.

[63] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.

[64] B. Terry, A. Demers, K. Petersen, M. J. Spreitzer, M. Theimer, and B. Welch. Session guarantees for weakly consistent replicated data. In

International Conference on Parallel and Distributed Information Systems, pages 140–149, September 1994.

[65] A. Vahdat, M. Dahlin, T. Anderson, and A. Aggarwal. Active Naming: Flexible Location and Transport of Wide-Area Resources. In *2nd USENIX Symposium on Internet Technologies and Systems*, October 1999.

[66] M. van Steen, P. Homburg, and S. Tanenbaum. The Architectural Design of Globe: A Wide-Area Distributed System. Technical Report IR-422, Vrije Universiteit, Amsterdam, March 1997.

[67] A. Venkataramani, R. Kokku, and M. Dahlin. TCP-Nice: A mechanism for background transfers. In *OSDI02*, December 2002.

[68] A. Venkataramani, P. Yalagandula, R. Kokku, S. Sharif, and M. Dahlin. Potential costs and benefits of long-term prefetching for content-distribution. In *Web Caching and Content Distribution Workshop*, June 2001.

[69] A. Whitaker, M. Shaw, and S. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *2002 USENIX Technical Conference*, June 2002.

[70] K. Worrell. Invalidation in Large Scale Network Object Caches. Master's thesis, University of Colorado, Boulder, 1994.

[71] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar. Engineering web cache consistency. *ACM Transactions on Internet Technologies*, 2(3), August 2002.

[72] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Volume Leases to Support Consistency in Large-Scale Systems. *IEEE Transactions on Knowledge and Data Engineering*, 11(2):563–576, February 1999.

[73] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using Smart Clients to Build Scalable Services. In *1997 USENIX Technical Conference*, January 1997.

[74] H. Yu and A. Vahdat. The costs and limits of availability for replicated services. In *18th ACM Symposium on Operating Systems Principles*, 2001.

[75] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *4th Symp. on Operating Systems Design and Impl.*, pages 305–318, Oct 2000.

Vita

Amol Pramod Nayate was born in Wheaton, Illinois on February 12, 1978. He attended the University of Utah from 1996-1999 and received an undergraduate degree in Computer Science and Engineering. He then attended the University of Texas at Austin and received a Master's of Arts degree in Computer Science in 2001 under the supervision of Dr. Mike Dahlin. He has worked on product development teams at Microsoft (1998), IBM Austin (1999), and at IBM TJ Watson research center (2001).

Permanent address: 301 west 39th st. #207
Austin, Texas 78751

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.